Radboud University Nijmegen

Faculty of Social Science

FlowCoder

A Model of Symbolic Thought Composition via Program Synthesis using Generative Flow

Thesis MSc Artificial Intelligence

Student Information

Surname:	Hommelsheim
First name:	Ron
Student number:	s1000522
E-mail address:	ron.hommelsheim@ru.nl
Course code:	SOW-MKI94 (45 EC)
AI Specialisation:	Cognitive Computing

Supervisor Information

Role:	Supervisor
Surname:	Thill
First name:	Serge
Institute:	Radboud University
E-mail address:	serge.thill@donders.ru.nl
Supervision type:	Internal

Acknowledgement

I would like to express my sincere gratitude to my supervisor for the numerous insightful discussions, indulging my every new excitement for new ideas. Their guidance and expertise have been invaluable in shaping my research and in enhancing my understanding. Additionally, I would like to show appreciation to my parents, to my sister, and to my friends for their endless patience, encouragement, and unconditional belief in me. Lastly, the unwavering support of my partner throughout this journey has been a constant source of motivation and inspiration, which I am greatly thankful for.

I am truly fortunate to have such a strong support network, and I am deeply appreciative of the roles each of them has played in my academic and personal development.

Abstract

Artificial Intelligence (AI) research has historically been polarized between symbolic reasoning, characterized by precise but inflexible models, and deep learning, known for its analogical reasoning capabilities yet limited by poor generalization to Out-of-Distribution (OOD) data and high data requirements. Despite the successes of Large Language Models (LLMs), challenges such as confabulation and a lack of causal reasoning persist, underscoring a gap in achieving human-like cognition. This study introduces FlowCoder, a novel program synthesizer that synergizes the symbolic and subsymbolic paradigms by embedding programs within a Transformer-based implicit world model and utilizing Generative Flow Network (GFlowNet) for program inference. FlowCoder demonstrates an ability in extrapolating to OOD tasks and performing efficient one-shot inference given sufficient training. This work showcases FlowCoder's capacity as a model in program synthesis and more so discusses its potential and theoretical implications as a model for a Language of Thought (LOT).

Contents

List of Figures

List of Tables

1 Introduction

Imagine your brain as an interactive game engine. Just as a game engine generates dynamic virtual environments, complete with rules and physics that players interact with, the brain constructs a model of the real world. This model includes rules (physical laws, social norms), entities (objects, people), and interactions (how things work and relate to each other). We learn to navigate and predict our environment, constantly updating our internal model based on new experiences and information. This analogy, introduced by Ullman *et al.* [80] extends beyond mere perception, encompassing imagination, dreams, and memory. Each of these cognitive functions can be seen as manifestations of the brain's ability to generate, manipulate, and explore various scenarios and possibilities within its internal model. Dreams and imaginative constructs, while seemingly detached from reality, are composed of the same 'material' as our waking perceptions – they are all products of the brain's simulation capabilities [65]. The self, in this view, becomes both a creator and a perceiver of its subjective reality, a reality that, while grounded in the external world, is ultimately shaped by the mind's interpretative and predictive faculties.

In the following, an overview of the fundamental concepts used in this thesis are presented, focusing on program synthesis and its relevance to understanding human cognitive processes. Strengths and limitations of current models are discussed before outlining the approach of overcoming said limitations. FlowCoder [1] is introduced as a proposed model for program synthesis. A computational model and implementational details are discussed. Two experiments are outlined and their results are analyzed. Finally, improvements and various implications of the model are highlighted.

1.1 Background

Traditionally, Artificial Intelligence (AI) research has been approached from two general directions. Good Old-Fashioned AI (GOFAI) is based on symbolic reasoning. Symbols have no internal structure but gain significance in relation to other symbols. Models based on formal reasoning are said to be precise and tend to generalize well, yet they are slow and inflexible. Instead, deep-learning relies on distributed vector representations that have a similarity structure and facilitate analogical reasoning, which may be a core function of cognition [6, 37]. These models tend not to generalize well to Out-of-Distribution (OOD) data and are notoriously data-hungry. Moreover, composition, systematic generalization (OOD), and abstraction are often argued to be crucial aspects of human cognition [10, 17, 26, 37, 50], which may be facilitated by a latent innate capacity for the representation and construction of part-whole hierarchies [8, 27, 34, 58, 70, 74].

Language of Thought Dehaene *et al.* [21] posit that human cognition is uniquely characterized by its ability to form symbolic representations and recursive mental structures akin to a Language of Thought (LOT), enabling the creation of domain-specific conceptual systems. This cognitive ability allows for the generation of new concepts through the compositional arrangement of existing elements, a process exemplified by the derivation of geometric concepts [1]. Cognition simplifies complex patterns into mental representations via mental compression, where the complexity of a concept is measured by the length of its mental representation as per the Minimum Description Length (MDL) principle.

[1] FlowCoder is available at `https://github.com/R1704/master_thesis`

To illustrate, when learning to play chess, rather than remembering as many games as possible, we capture the few rules, through which we can understand and explain all instances of the game.

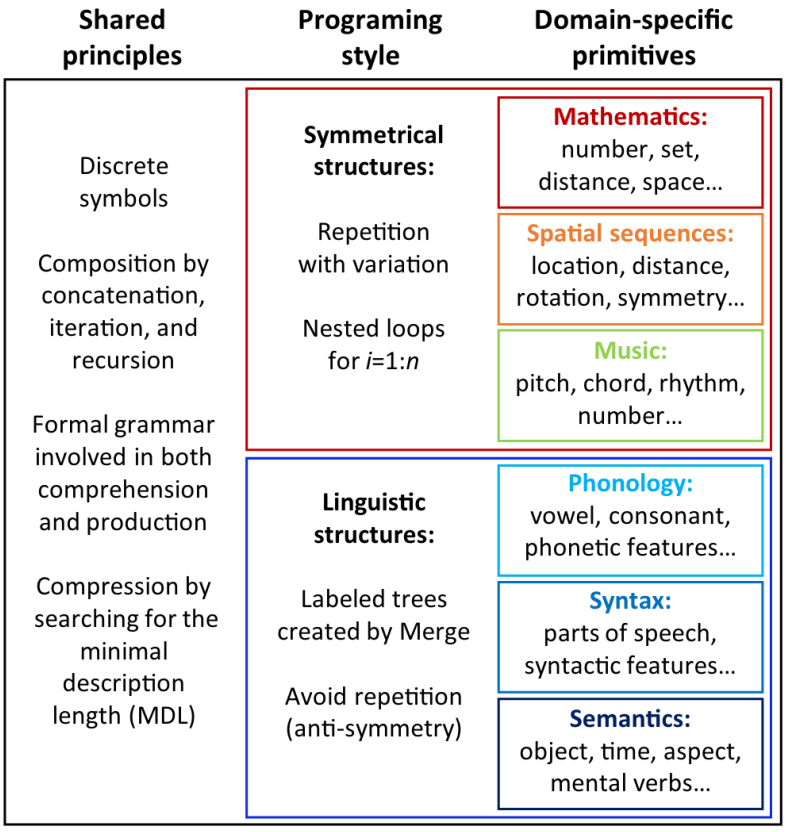

Figure 1: Human cognition is underpinned by multiple mental Domain Specific Languages (DSLs). Each language has basic building blocks - primitives which can be programmatically composed to form more complex structures. Dehaene *et al.* [21] distinguish between symmetric and asymmetric programming styles. The design principles of these mental languages are shared. They are symbolic, recursive, compositional, use formal grammar, and compress programs by adhering to the minimal description length principle. The diagram was taken from the original paper [21].

Current versions of the LOT posit that the brain implements mechanisms analogous to those found in probabilistic programming languages, enabling it to represent and infer the probabilistic structure of the world [49, 71]. A program here can be thought of a procedure that generates more examples of the same concept. If a program would represent the concept "animal", it would generate examples such as "giraffe", "zebra", "fish", and so on. Higher-level programs could produce lower-level programs. In this paradigm, the essential aspect of compositionality gives rise to a part-whole hierarchical structure, which facilitates systematic generalization.

Program Synthesis and Problem Statement This computational model of cognition can be formalized as *program synthesis*, where the goal is to automatically construct programs that satisfy a given set of specifications. Program synthesis involves defining a domain-specific language with a set of primitives and rules, and then searching within this language for a program that satisfies a given set of input-output relations, representing the task at hand. This process is fundamentally about mapping a defined task to an executable program within the constraints of the specified

DSL.

A Domain-Specific Language $\mathcal{D}$ is defined as a set of syntactic and semantic rules that determine the structure and meaning of valid expressions in the language. Formally, a DSL can be represented as:

$$\mathcal{D} = \{\mathcal{S}, \mathcal{O}, \mathcal{R}\}$$

where $\mathcal{S}$ is the syntax defining the structure of valid expressions, $\mathcal{O}$ is the set of operations (or primitives) available in the language, and $\mathcal{R}$ are the semantic rules that assign meaning to the expressions.

Primitives in the DSL are the basic operations from which programs are constructed. Each primitive $o \in \mathcal{O}$ can be thought of as a function:

$$o : A \to B$$

where A is the set of input types and B is the output type for the primitive.

A task $x \in X$ in program synthesis is defined as a set of input-output pairs that specify the desired behavior of a program. Formally, a task can be represented as:

$$x = \{(x_{in_1}, x_{out_1}), (x_{in_2}, x_{out_2}), ..., (x_{in_n}, x_{out_n})\}$$

where each pair (x_{in_i}, x_{out_i}) consists of an input x_{in_i} and the corresponding desired output x_{out_i}. The objective of program synthesis is to find a program ρ within the language $\mathcal{D}$ that satisfies the task x. Formally, this can be seen as a search problem: Find $\rho \in \mathcal{D}$ such that for every $(x_{in_i}, x_{out_i}) \in x$, $\rho(x_{in_i}) = x_{out_i}$.

DreamCoder DreamCoder (DC) stands out as a particularly effective model in program synthesis, creating programs from basic primitives and tasks with the goal of developing its own domain-specific language [24]. It employs an adapted wake-sleep algorithm, initially introduced by Hinton *et al.* [36], to simultaneously train a generative model and a recognition network. The generative model is tasked with learning a probability distribution across programs, while the recognition network is designed to map tasks to specific programs, facilitating a neurally-guided exploration of the program space. This process leverages the recognition network to implement a parallel search strategy, blending best-first and depth-first searches to prioritize programs based on their probabilities.

The model significantly narrows the search scope by abstracting frequently used sub-routines into more readily accessible concepts, thereby enhancing scalability. This abstraction not only reduces the depth of the search tree but also limits its breadth, with the abstraction phase playing a pivotal role in refactoring subroutines in accordance with the Minimum Description Length principle and in the learning of the Domain Specific Language.

The tasks addressed by DreamCoder can either be generative, such as image creation, or conditional, like establishing input-output relationships for list sorting. Examples of tasks from various domains are depicted in Figure 2(A), while Figure 2(B) illustrates the process of learning to sort a list. The figure shows initial primitives on the left, a middle section highlighting the library of

learned concepts and the established part-whole hierarchy, and on the right, the ultimate solution employing `concept15`, which itself incorporates previously abstracted concepts.

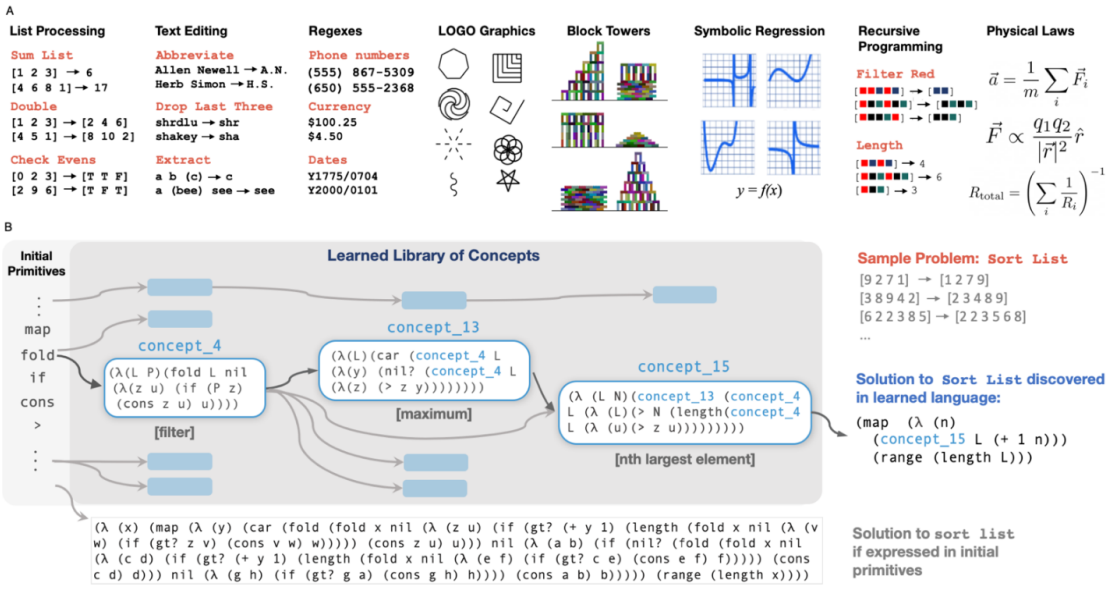

Figure 2: (A) Tasks across eight distinct domains. (B) Illustration of the concept library that has been acquired. The left side displays the foundational primitives that are used to construct the concepts shown in the central area. To the right, a task is presented through input-output relationships alongside the derived solution. Below, this solution is reformulated using solely the initial primitives. Image taken with permission from the original paper [24].

DeepSynth Fijalkow *et al.* [25] propose a framework called "distribution-based search", in which they investigate the difficult problem of searching through a DSL to find programs matching a specification in a vast hypothesis space. They introduce DeepSynth [2], a general-purpose program synthesizer which constructs programs from input-output examples, and a useful framework allowing us to test different models and search methods, which I am using in this project. The authors discuss different program finding strategies. Specifically, they find that both enumerative search (as in DC) and sampling are viable strategies, where search is associated with prioritizing quantity, i.e. creating many programs quickly, whereas sampling strategies prioritize quality but may be slower, since resampling may occur. An additional benefit of sampling over search is space efficiency - already created programs don't need to be memorized. Here, an initial DSL along with suitable syntactic constraints compile into a Context Free Grammar (CFG), defining the possible structures of programs within its DSL. A CFG consists of a set of production rules that describe how to generate strings from a set of non-terminal and terminal symbols. It is "context-free" because the production rules are applied regardless of the surrounding symbols. In DeepSynth, a prediction model is used to predict weights for a Probabilistic Context Free Grammar (PCFG), extending the CFG by associating probabilities with the production rules. This allows the grammar to not only generate the syntactic structure of a program but also to represent beliefs about the relative plausibility or frequency of different structures [3]. This is similar to DreamCoder's prior, consisting of a library of sub-routines combined with a weight vector. The PCFG guides the search

[2] `https://github.com/nathanael-fijalkow/DeepSynth`
[3] See appendix Appendix D for a formalization of CFGs and PCFGs.

and inference process towards more likely programs. DreamCoder however, does not specifically use a PCFG. Both frameworks employ a typed λ-calculus, hence there are restrictions on program arguments, etc. (syntactical constraints). DreamCoder performs type inference during program generation. To spare computational cost, DeepSynth constructs the CFG beforehand which in turn increases its size. Fijalkow *et al.* [25] compare different search strategies and show that methods that do not use a machine-learned PCFG (e.g. Depth First Search (DFS)) barely solve any tasks, demonstrating the necessity for better strategies.

1.2 Limitations

Although DreamCoder and DeepSynth prove to be successful in synthesizing programs, their methods reveal a foundational limitation: their heavy reliance on syntactical constraints. While these constraints are undoubtedly vital for ensuring the correctness of generated programs, they do not necessarily guarantee a deep understanding or utilization of semantic relationships within the code. Additionally, Kim *et al.* [46] explain that associating only a scalar per rule misses a lot of information. A distributed representation of the DSL would therefore be beneficial. We could imagine a program space in which certain symmetries could be leveraged. One could argue e.g. that "+" is to "−" as "÷" is to "×". These semantic relationships may be missed in the previously discussed models.

1.3 Approach

Transformers and Self-Attention The Transformer architecture, originally introduced in 2017 by Vaswani *et al.* [82], has proved to be widely successful in a wide range of applications [45, 86]. Transformers use self-attention, a mechanism that enables dynamic selection and focus on specific parts of the input, as opposed to treating all parts equally. It effectively allows the network to "attend" to, or give more weight to, certain inputs over others during the processing stage. The self-attention mechanism allows for an understanding of not just the structural arrangement of elements in a sequence (syntax) but also their deeper, contextual relationships (semantics) [87]. In this thesis I will use this model for a rich representation of programs. However, training the Transformer is difficult from only a few examples. Therefore, I will combine the approach with an amortized sampler, explained in the following.

GFlowNet Generative Flow Networks (GFlowNets), introduced by Bengio *et al.* [5], are a class of generative models designed to learn to construct compositional objects from a target distribution over complex high-dimensional spaces, particularly where explicit density estimation is challenging and diverse candidates are encouraged. GFlowNets learn a stochastic policy for generating sequences of actions that lead to the construction of a sample. The model generates sequences of actions that build a sample, with the generation frequency of each sample being proportional to an associated reward function. In other words, GFlowNets are applicable in problems where complex structures are composed from simple building blocks and have been used in molecular composition from atoms [5], in grammar induction [38], and in Bayesian structure learning [22]. The learnt policy becomes an amortized sampler. This means that the extensive training invested in the model results in a system capable of efficiently generating new samples without the need for additional, extensive computation for each new instance. Moreover, the model can be used for offline training, i.e. from data that is not from the observed distribution. This aspect is crucial for

OOD generalization and may be the remedy for data-hungry Transformers.

1.4 Research Question, Aim, Motivation

In recent advancements, State-of-the-Art (SOTA) models like DreamCoder have demonstrated proficiency in program synthesis. However, they often lack a semantically rich state representation and heavily rely on search algorithms for constructing programs. This thesis aims to investigate a novel approach by combining the strengths of two distinct architectures: the Transformer and GFlowNet. The Transformer architecture is known for its ability to learn rich state spaces, albeit with a significant data requirement and limited generalization to Out-of-Distribution tasks. On the other hand, GFlowNet, with its capability for amortized sampling, presents a promising solution to overcome these challenges. The central hypothesis of this thesis is that the integration of these two architectures could yield a powerful program synthesizer. This synthesizer would be capable of solving tasks with minimal examples, specifically in the list-editing domain. Furthermore, theoretical and computational challenges are identified and addressed within the realm of neural program synthesis.

This research explores the potential alignment of the proposed model with the Language of Thought hypothesis, suggesting a programming language-like mental representation underpinning human thought. Thus, this research not only aims to address a practical gap in program synthesis but to explore the role of program synthesis in a model of cognition, thereby contributing to the philosophical and psychological understanding of thought, and intelligence.

1.5 Scope and Limitations

The concept of abstraction in program synthesis is necessary for the model to learn its own DSL. Abstraction effectively narrows the depth of the search tree through program refactoring and identifies common patterns, thereby aiding in generalization. Additionally, abstraction is essential in optimizing for the Minimum Description Length, which is a useful inductive bias humans seem to employ [72]. However, in this research, abstraction was not implemented. This decision was primarily guided by time constraints. As a result, I focus on modeling a program synthesizer that solves tasks and on testing its abilities, rather than additionally learning the DSL. Consequently, it is anticipated that the model will not optimize for parsimonious programs.

2 Methods

In the following, the foundational concepts used within this thesis are outlined in detail. The proposed model, FlowCoder, is introduced and an exact computational model is given, before describing implementational details and training and testing strategies.

2.1 Foundations & Computational Framework

GFlowNet GFlowNets create a Directed Acyclic Graph (DAG) over the state space, where vertices correspond to states or partial samples and edges denote transitions or adding a component to a partial sample, in which the edges carry a flow from source to targets [5]. In GFlowNets, the "flow" in the network corresponds to the process by which the network constructs a sample, which can be thought of as a path in a graph where nodes are partial samples, and edges correspond to adding a component to the partial sample. The core training objective for a GFlowNet is to satisfy the flow matching constraint. The idea is to ensure that the flow into any state (a partially constructed sample) matches the flow out of it, given the reward associated with complete samples. The flow here refers to the expected transitions into or out of a state under the model's stochastic policy.

Formally, a state s represents a partial object a certain stage in the generative process. A trajectory τ is a sequence of states $s_0, s_1, ..., s_T$; the model traverses from an initial state s_0 to a terminal state s_T, where the target structure is complete. A trajectory τ is formed by a sequence of actions $a_1, a_2, ..., a_T$, where each action a_t transitions the model from state s_t to state s_{t+1}. The sequence of actions is governed by a policy π, which defines the probability of choosing a particular action given the current state. The flow $F(\tau)$ of a trajectory τ is defined as the product of the probabilities of each transition along the trajectory, multiplied by the reward $R(s_T)$ of the terminal state, normalized by a partition function Z.

$$F(\tau) = \frac{R(s_T)}{Z} \prod_{t=0}^{T-1} \pi_\phi(s_{t+1}|s_t) \tag{1}$$

The partition function Z ensures that the sum of flows over all possible trajectories equals one, effectively normalizing the distribution. Since we don't know Z, we can estimate it by parameterizing it as Z_θ. The flow matching constraint enforces that for any given non-terminal state s, the total flow into s must equal the total flow out of s:

$$F(\tau) = F(\tau') \tag{2}$$

where $F(\tau')$ is the reverse trajectory. The partition function is estimated and utilized to balance the probabilities throughout the DAG. Following Malkin *et al.* [56], we can utilize this property to create a suitable loss function - Trajectory Balance (TB) Loss to train the GFlowNet.

$$\mathcal{L}_{TB} = \left(\log Z_\theta(x) + \sum_{t=0}^{T-1} \log \pi_\phi(s_{t+1}|s_t, x) - \log R(s_T|x) \right)^2 \tag{3}$$

Here s_T is a constructed object, in this case a program ρ. The complete derivation of the TB loss can be found in Appendix F.
To compute the reward $R(\rho|x)$, the sampled program ρ is executed and evaluated to get $\rho(x_{in}) = \tilde{x}_{\text{out}}$ after which the output pair $(x_{\text{out}}, \tilde{x}_{\text{out}})$ is compared.
Additionally, a state-task representation is encoded by $T_\theta(x, s_t)$, parameterized by θ. In this project, T will be referred to as state-task representation or generative model interchangeably. The output $z = T_\theta(x, s_t)$ is the task-state representation which policy π takes as an input.

Expectation-Maximization (EM) Expectation-Maximization is an algorithm for finding maximum likelihood estimates in models with latent variables. It iteratively optimizes a lower bound on the likelihood of the observed data, alternating between inferring the most likely latent states (E-step) and optimizing model parameters given these states (M-step) [32]. EM has been used repeatedly in in the realms of grammar induction or program synthesis. DreamCoder uses a generalized version of EM [24]. Kim *et al.* [46] use EM for grammar induction and Hu *et al.* [38] extend Kim *et al.*'s [46] methodology, and propose a novel method GFlowNet-EM. Similar to Hu *et al.* [38], I am updating the two models separately.

In the E-step a program ρ is sampled and the parameters of policy π_ϕ are updated using the Trajectory Balance Loss (Equation 3). Here, the policy tries to find better programs solving the task.
The M-step comprises sampling a program ρ and updating the model parameters T_θ with reward $-\log R(\rho|x)$, where the goal is to refine the generative model.

Sleep Inspired by DreamCoder and GFlowNet-EM, I am employing a modified wake/sleep algorithm, originally introduced by Hinton et al. [36].

In *Replay*, the forward policy is trained on previously solved task-program pairs (x, ρ), using the trajectory τ to guide the model to the correct solution and optimizing on the forward logits. Here x is sampled from the empirical distribution and ρ is sampled from the forward policy π_ϕ. Additionally, a sleep weight γ is applied to strengthen the gradient. Formally:

$$\nabla_\phi \mathcal{L}_{\text{Replay}} = \mathbb{E}_{x\sim X, \rho\sim\pi_\phi(.|x)} \left[-\gamma \cdot \log \pi_\phi(\tau|x, \rho)\right] \tag{4}$$

If a correct solution has been found during the E-step, I immediately let the model train on these trajectories of correct solutions so as to consolidate these. After the E-step I again train the model on a set (in the mathematical sense, meaning no duplicates) of all the correct solutions, so that it doesn't forget solutions to other tasks. Replay is applied stochastically, given the hyperparameter ξ.

During *Fantasy*, the policy is trained on hypothesized programs with tasks from the empirical distribution. Programs are executed using task *inputs* from the empirical distribution to output a predicted *output* and thus create correct task-program pairs $(\tilde{x}, \rho)$. Programs that produce unwanted behavior such as producing constants (that are not dependent on the task), or NaNs are filtered out. Then, similarly to the methodology of Replay, the model is trained on these pairs. Formally:

$$\nabla_\phi \mathcal{L}_{\text{Fantasy}} = \mathbb{E}_{x\sim X, \tau\sim\pi_\phi(.|x)} \left[-\gamma \cdot \log \pi_\phi(\tau|\tilde{x}, \rho)\right] \tag{5}$$

Fantasy is also applied stochastically, given the hyperparameter σ. Incorrectly proposed programs or randomly generated programs allows the model to learn which task-program pairs do or do not make sense, while fantasy on correct programs allows the model to generalize programs to different input-output pairs.

Optimization Techniques Hu *et al.* [38] propose several optimization techniques that I adopted in this project. These will be described in the following.

E-step Loss Thresholding Rather than training the GFlowNet to a loss of zero after each M-step, we can apply a linearly decreasing moving average loss δ as a threshold to trigger the M-step using the hyperparameter α, to save computational cost. Here I use the recursive formula:

$$\delta = \alpha \cdot \mathcal{L}_{TB} + (1 - \alpha) \cdot \delta \tag{6}$$

Exploration Since we want to find many modes in the E-step and want to avoid getting stuck in local optima, several exploration techniques can be employed. The hyperparameter $\{\beta|\beta \in \mathbb{R}_{[0,1]}\}$ can be used to exponentiate the forward policy: $\pi_\theta(s_{t+1}|s_t)^\beta$. Moreover, ϵ-uniform sampling can be used to the deter the model from repeating known routes by mixing the predicted logits with a uniform distribution. ϵ is chosen to be $\{\epsilon|\epsilon \in \mathbb{R}_{[0,1]}\}$.

2.2 FlowCoder

Program Representation DeepSynth represents programs as Abstract Syntax Trees (ASTs). An AST is a tree representation of the syntactic structure of the program, with nodes representing operations or primitives and edges representing their compositional relationships. Other common methods in program synthesis are used such as deBruijn indexing, which is a technique to represent bound variables in a way that avoids naming conflicts and simplifies variable substitution [20]. Named variables are replaced with numeric indices representing the number of enclosing λ-abstractions that bind the variable. See Figure 3 for a visualization of an AST. Moreover, the authors implement a polymorphic type system which facilitates the constraint of compiling a CFG that produces programs of a desired type-request [4].

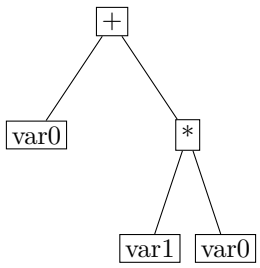

Figure 3: An example of an abstract syntax tree (AST). This translates to the function `f = var0 + var1 * var0`.

Generative Model To embed programs effectively within a neural network, it is essential to represent them in a format that is compatible with the neural network's architecture. Abstract syntax trees, which are inherently 2-dimensional and include information of parent, children and

[4]The DeepSynth implementation can be found at `https://github.com/nathanael-fijalkow/DeepSynth`

sibling nodes, should be translated into a neural network format while ensuring the preservation of this valuable structural information. Various concepts have emerged, including using Graph Neural Networks (GNNs), see e.g. [2, 9, 39, 83]. Others proposed special Tree-Transformers which are able to encode ASTs, see e.g. [66, 84]. He *et al.* [33] find that standard Transformers achieve similar results to Transformers where tree position information is explicitly encoded. This is presumably because of the positional encoding, which is a way of adding fixed sinusoidal functions with different frequencies and phases to the embeddings of tokens in a sequence, enabling the neural network to discern the position of each tokens through these distinctive patterns; and more importantly, because of the fundamental component of the Transformer, which is the self-attention mechanism, formalized as [82]:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{7}$$

This mechanism, through the query (Q), key (K), and value (V) matrices, allows the model to dynamically assign importance to different parts of the input sequence. The scaling factor $\sqrt{d_k}$ normalizes the dot product, aiding in stabilizing gradients during training. Moreover, the Transformer employs Multi-Head Attention, enabling the model to concurrently process information from different representation subspaces, enhancing its capability to capture diverse features. Therefore, I decided to use the standard Transformer architecture, which takes a 1-dimensional sequence as an input, meaning the AST has to be linearized.

Rather than embedding the primitives of the DSL to construct ASTs, the already preprocessed rules of the context-free grammar are embedded, created in conjunction with the syntactic constraints, thus essentially predicting edges rather than nodes. The array of CFG rules has to be converted back to AST format for evaluation, which can be easily done within the DeepSynth framework. In a plausible model of cognition, we don't possess an explicit representation of the CFG but rather infer it. However, in the case of DeepSynth, it generates a more extensive CFG instead of conducting type inference, which is the approach employed in DreamCoder. Consequently, a lookup to identify the syntactically permitted rules must be performed, thereby filtering out those that do not conform to the syntax. Naturally, in principle the model could be allowed to generate syntactically incorrect programs and assign them a reward of 0 during evaluation, but to expedite the learning process, I refrain from doing so.

The RuleEncoder begins by collecting rules, which are pairs of non-terminals and corresponding program actions, from the CFG and adds special tokens such as 'PAD' (padding), 'START' (sequence start), and 'STOP' (sequence end). Each rule is passed through an embedding layer. During the forward pass, batches of state sequences (each state sequence, or trajectory τ, representing a series of CFG rules) are processed. To ensure uniformity across different sequences within a batch, padding is applied.

The IOEncoder is tasked with encoding Input-Output (IO) pairs into continuous vector representations. Each input-output pair is tokenized using a predefined lexicon (a list of symbols representing the possible range of inputs and outputs), which includes special tokens such as 'PAD' (padding), 'IN' (input start), and 'OUT' (output start). This tokenization is critical for distinguishing between different parts of the input-output pairs. The tokenized input-output pairs are concatenated into a single sequence, with the 'IN' and 'OUT' tokens demarcating the transition from input to output. Padding is then applied to ensure that all sequences have

the same length, aligning them to the maximum allowed size determined by `n_examples_max` (the maximum number of input examples that can be processed) and `size_max` (the maximum size of elements in a list). These parameters have been adapted from DeepSynth. The padded sequences are passed through an embedding layer. This embedding is crucial for capturing the semantic relationships between different tokens.

The Transformer initializes with the IOEncoder and the RuleEncoder. Positional Encoding is applied to the output of both encoders. This step is vital as it adds information about the sequence order to the model, allowing the Transformer to interpret the sequence data effectively. The Transformer employs two types of masks: padding masks for IO sequences and square subsequent masks for state sequences. The padding mask ensures that the model does not process padding tokens, while the square subsequent mask prevents positions from attending to subsequent positions, maintaining the autoregressive property in the generation process.

Forward Policy At each step, the forward policy takes the Transformer output as an encoded state and predicts log probabilities over the CFG rules, after which softmax is applied to get a distribution between 0 and 1 and sample the next action. The forward policy is implemented as a Multi-Layer Perceptron (MLP) and predicts forward logits from the Transformer's output, guiding the generative process.

Partition Function The partition function Z_θ, which is an estimation of the sum of all rewards $R(\rho|x)$, serves as a normalizing factor, ensuring that the probabilities generated by the model are well-calibrated and interpretable. It is implemented as a MLP layered on top of the Transformer, outputting a scalar.

See Table C.1 for the exact parameterization of the models.

Sampling Programs In the process of constructing an Abstract Syntax Tree, there exists flexibility in the order of expansion. Nodes can be expanded in a depth-first, breadth-first, etc. manner, or for instance, one can adopt a bottom-up approach, wherein terminal nodes are predicted initially and subsequently connected in a progressive manner. This approach offers the advantage of enabling the evaluation of partial expressions, which, in turn, can serve to inform the model and enhance computational efficiency, albeit at the expense of increased memory requirements. Alternatively, we may consider employing a model akin to the forward policy, predicting the subsequent node (or edge) to be expanded. However, I chose to sample the actions for the tree construction in a depth-first manner, mainly for two reasons. First, for simplicity, and second so that the AST when linearized always has the same order, potentially giving the Transformer a useful inductive bias.

Specifically, the state of each program in the batch is initialized with a 'START' token, representing the initial state of the program generation process. A frontier, implemented as a queue, is initialized for each program to manage the sequence of non-terminals that need to be expanded. This method has been adapted from DeepSynth to fit with the FlowCoder framework and mechanisms. The core of the method is a loop that continues until all frontiers are empty, indicating that all programs in the batch have been fully generated. Within this loop, the model computes logits and partition functions. The sampling method includes an exploration mechanism, where with a

probability ϵ, uniform sampling is used instead of the model's logits. This exploration is crucial for introducing variability and avoiding local optima in the generation process. Additionally, tempering is applied to logits using a factor β, modulating the sharpness of the probability distribution used for sampling. For each program in the batch, the method iteratively samples a rule based on the current non-terminal and updates the program's state and cumulative logits. This process involves creating a mask to block invalid actions, applying the mask to logits, and sampling an action (rule) based on the masked logits. Each sampled rule is decomposed into a non-terminal and program component, updating the program's current state and expanding the frontier with the rule's arguments. Once all frontiers are empty, the final programs are reconstructed from their compressed representations, using methods provided by DeepSynth.

Reward In order to train the model, a reward function has to be operationalized. Various approaches exist for this purpose, the simplest being a binary reward, as employed in DreamCoder. Does the output of the program match the actual output or not? A binary reward however, is not very informative. A reward that provides a gradient is much more useful. Bengio *et al.* [7] propose an Energy-Based Model (EBM), wherein the model learns to associate favorable outcomes with low energy states and unfavorable outcomes with high energy states. I found that this method introduced unnecessary complexity to the model. Instead, the Levenshtein edit distance was used [5], which is a measure of the similarity between two strings [52]. Specifically, it quantifies the minimum number of single-character edits (i.e., insertions, deletions, or substitutions) required to transform one string into another. See the appendix (Appendix E) for an in depth formalization of the metric. Since the Levenshtein distance returns a discrete value, it was normalized over the maximum length of the sequences, and since there may be more than one example per task, it was averaged over all examples. Moreover, a maximum reward parameter was applied, to scale a correct solution up and give the model a stronger gradient. In my experiments a maximum reward of 10 was used.

2.3 Design

In the following section I will elaborate on the exact training and test methods employed, including hyperparameters. The tasks that were used are input-output relations in the list editing domain, originally from DreamCoder, see Table 2 for examples. These were filtered given the syntactic constraints (e.g. type, lexicon, etc.) and provided by DeepSynth. In their paper, Fijalkow *et al.* [25] discuss that some of the tasks are impossible to solve given the DSL, therefore, those were additionally filtered out, finally leaving 95 tasks in the dataset. Tasks can be of a similar variety (e.g. `add-k with k=1` and `add-k with k=2` belong to the same group). There are 25 such task groups in total. Each task can have between 1 and 15 examples. A task is considered solved if a program solves all examples of the task [6]. The DSL is essentially a dictionary of primitive types and semantics in a typed λ-calculus, including functions like `index`, `car`, `append` as well as numerical functions like `+`, `*`, `mod`, `is-prime`, etc., all written in Python (Appendix A). Table 1 describes the syntactical constraints used in the experiment, which were mostly adapted from DeepSynth and DreamCoder.

[5] The implementation of the Levenshtein distance I used can be found at `https://github.com/maxbachmann/Levenshtein`

[6] All tasks can be found in my GitHub repository.

Parameter	Value	Description
`type`	`list(int)` $\rightarrow$ `list(int)`	The input as well as the output should be a list of integers, so the CFG should reflect that, filtering rules that do not conform to that criterion.
`lexicon`	$[-30, 30] \in \mathbb{Z}$	The lexicon is a uniform distribution of integers and in the range from -30 to 30
`maximum argument number`	1	This is the maximum number of arguments a function could have.
`size max`	10	The maximum number of values in a list
`max number of examples`	15	The maximum number of examples per task

Table 1: Syntactical Constraints

All experiments were trained in random order with a batch size of 4 where each task in the batch is the same. This was done to avoid a credit assignment problem while increasing efficiency. Since the forward logits are summed up and averaged over for all trajectories, training the model at multiple task at once might have confused which trajectory was rewarding. However, batching, especially when training on a GPU has computational benefits. Furthermore, each task is trained for 5 epochs with 2.000 E-steps and 2.000 M-steps in each epoch. The moving average threshold (see Equation 6) δ was initialized with 150 and linearly decreases using the hyperparameter α set to 0.3. The exploration parameters β and ϵ were set to 0.7 and 0.3, respectively. The sleep weight γ was set to 10 on all experiments. The learning rates for the generative model as well as for the forward policy were set to 0.0001. During inference I ran the model for 100 steps. Since the batch size is 4, this creates 400 programs per task. A table of all hyperparameters can be found in Appendix B. The experiments were run on a single NVIDIA Tesla T4 GPU. Throughout the five epochs, each consisting of 2.000 E- and M-steps with a batch size of four, the model generated approximately 80.000 programs per task.

Task	Input	Output
`remove gt 2`	`[1,2,7,5,1]`	`[1,2,1]`
`caesar-cipher-k-modulo-n with k=5 and n=4`	`[2, 2, 0, 1, 2, 3, 3]`	`[3, 3, 1, 2, 3, 0, 0]`
`prepend-index-k with k=3`	`[15, 12, 9, 14, 7, 9]`	`[9, 15, 12, 9, 14, 7, 9]`
`add-k with k=4`	`[16, 10, 7, 12, 13, 3]`	`[20, 14, 11, 16, 17, 7]`
`append-k with k=2`	`[1, 5, 15]`	`[1, 5, 15, 2]`

Table 2: Task Examples

Task	Program
remove gt 2	(filter (gt? 3) var0)
caesar-cipher-k-modulo-n with k=5 and n=4	(map (mod 4) (map (+ 5) var0))
prepend-index-k with k=3	(cons (index 2 var0) var0)
add-k with k=4	(map (+ 4) var0)
append-k with k=2	(append 2 var0)

Table 3: Examples of tasks and programs solving the tasks.

3 Results

3.1 Experiment 1

This experiment tested the model's ability to generalize to unseen tasks. I set the program depth to 4 and trained on a random selection of only 17 out of 95 tasks. I set both parameters ξ as well as σ to 1, increasing the sleep frequency. Replay and fantasy are applied within each E-step, whenever a correct solution is found as well as after each E-step. All correct task-program pairs are saved and trained on after each E-step. The training took approximately one week. The model solved 15 tasks during training and 33 during inference. Notably, 8 tasks solved in training were not solved during inference, suggesting the model occasionally lost correct solutions found during training. During inference, the model therefore solved 26 tasks it had not seen before. Out of these, 7 were from previously unobserved task groups, demonstrating the model's capacity for generalization.

3.2 Experiment 2

In this experiment I want to examine the model's frequency of resampling, whether the model proposes varied solutions, and analyze its efficiency. Moreover, I analyze whether the EM cycles prove to be useful.

In this experiment I trained the model on a random 50/50 train-test split, which took about 3 days. All programs can be solved with a maximum program depth of 3, so I limited the model to that depth. This was merely done for efficiency. Moreover, ξ and σ were set to 0.3 and 1, respectively, and only a set (meaning no duplicates) of correct task-program pairs were saved and trained on, limiting the model's exposure to already correctly solved tasks [see sec for discussion on sleep]. The model successfully solved 33 out of the 48 tasks it was trained on. Although the model was trained on half of the tasks in the dataset, it does not generalize as well as FlowCoder in the first experiment. Sleep seems to be a crucial aspect of the algorithm.

As depicted in Figure 4, an average of about 11.000 unique programs were created per task, which is about 14% of the total programs created per task. This data also reveals certain task groups that posed more significant challenges than others. For instance, none of the tasks in the `caesar-cipher-k-modulo-n` group were solved, whereas all tasks in the `prepend-index-k` or `mult-k` groups were successfully completed. Refer to Table 2 and Table 3 for examples and correct solutions, respectively.

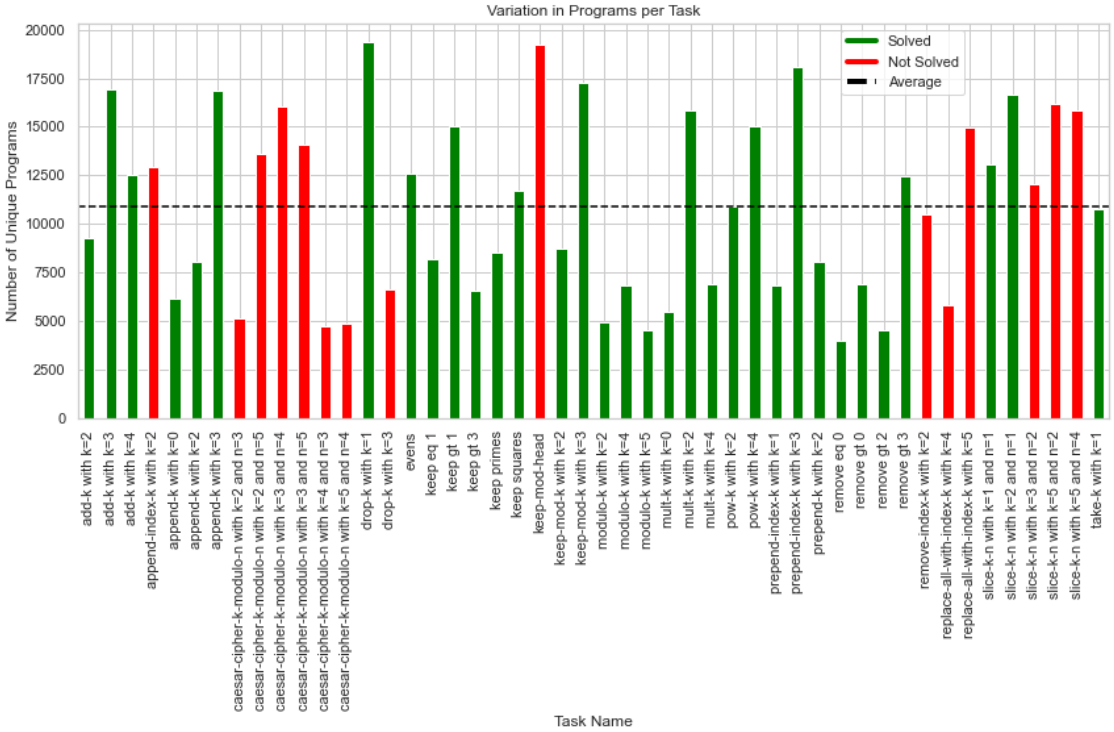

Figure 4: Bar plot of unique programs created per task. The sorted tasks the model has been trained are on the x-axis and the number of unique programs that have been created per task are on the y-axis. Bars of tasks that have been solved are coloured green and unsolved tasks are coloured red. The black dotted line demarcates the average number of uniquely created programs.

During inference, the model attempted to solve all 95 tasks sequentially, producing 400 programs in about 100 steps, taking roughly 30 seconds for each task. FlowCoder solved 42 tasks, of which 13 were unseen during training, as shown in Figure 5. 11 of these tasks belong to groups where at least one task was solved during training, e.g. the model was trained on (and solved) tasks `append-k` with `k=0, k=2, k=3` and during inference the model solved tasks `append-k` with `k=1, k=4, k=5`, which were previously unseen. 2 tasks were solved during inference without any precedent of tasks of a similar task group being seen in training. Additionally, all tasks solved exclusively during inference had not been exposed to the model in the training phase. The distribution of unique solutions also reveals that multiple tasks, irrespective of their exposure during training, exhibited a diverse range of solutions.

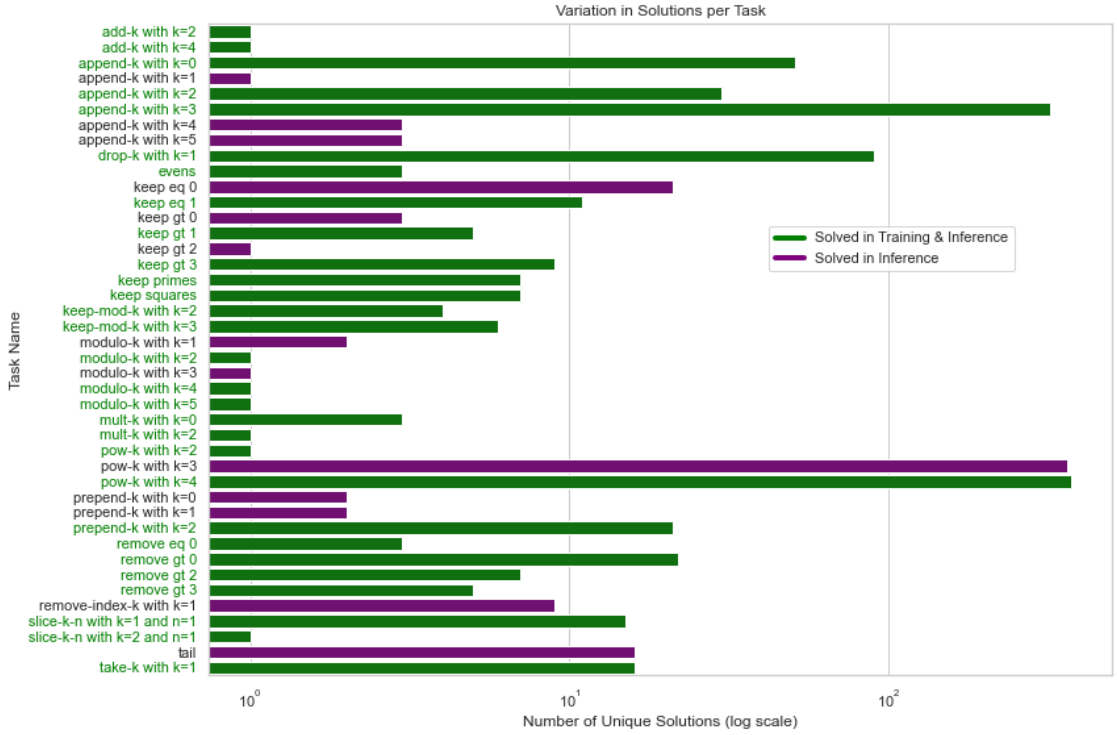

Figure 5: Distribution of unique solutions per task during inference, displayed in a log-scaled bar chart. The y-axis lists the task names, while the x-axis quantifies the number of unique solutions. The chart reveals that 13 tasks not solved in training were resolved during inference, with 11 belonging to groups with at least one task previously solved in training. Only solved tasks are shown.

In Table 4 we can see an example of varied solutions to a task. The model has found the shortest solution but also alternate solutions of the same task.

Task	**Solution**

`append-k with k=2`	`(append 2 var0)`
	`(append (mod 4 2) var0)`
	`(append (min 4 2) var0)`
	`(append (mod 5 2) var0)`

Table 4: Multiple found solutions.

On average, FlowCoder efficiently solves a task within approximately 8 steps. Notably, as illustrated in Figure 6, around half of the tasks are successfully resolved on the initial attempt. The model is able to rapidly solve many tasks that it had not encountered previously, often requiring fewer steps than the average. This suggests that when trained to convergence, the model may act as an efficient sampler.

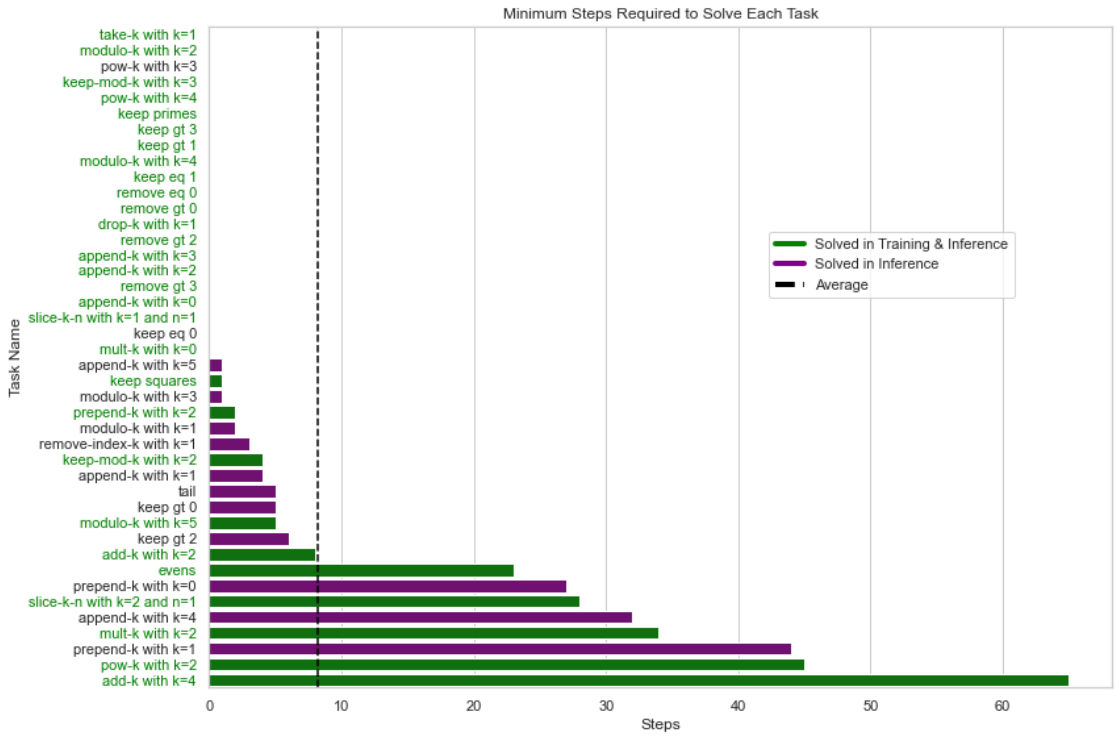

Figure 6: Analysis of the minimum number of steps required to solve tasks during inference. The x-axis represents the number of steps, and the y-axis lists the task names, sorted by the number of steps taken to find a solution. The dotted line indicates the average number of steps needed. Tasks solved both during training and inference are highlighted in green, whereas tasks exclusively solved during inference are in purple. Only solved tasks are shown.

An examination of the model's improvement across consecutive Expectation-Maximization (E-M) cycles is shown in Figure 7. The plot shows a clear upward trend in the average cumulative number of solutions, suggesting progressive improvement in both the forward policy and the generative model. The results of experiment 1 (subsection 3.1) are similar. As expected, the generative model during the M-step performs better than the GFlowNet during the E-step, since the E-step includes exploration while the M-step does not.

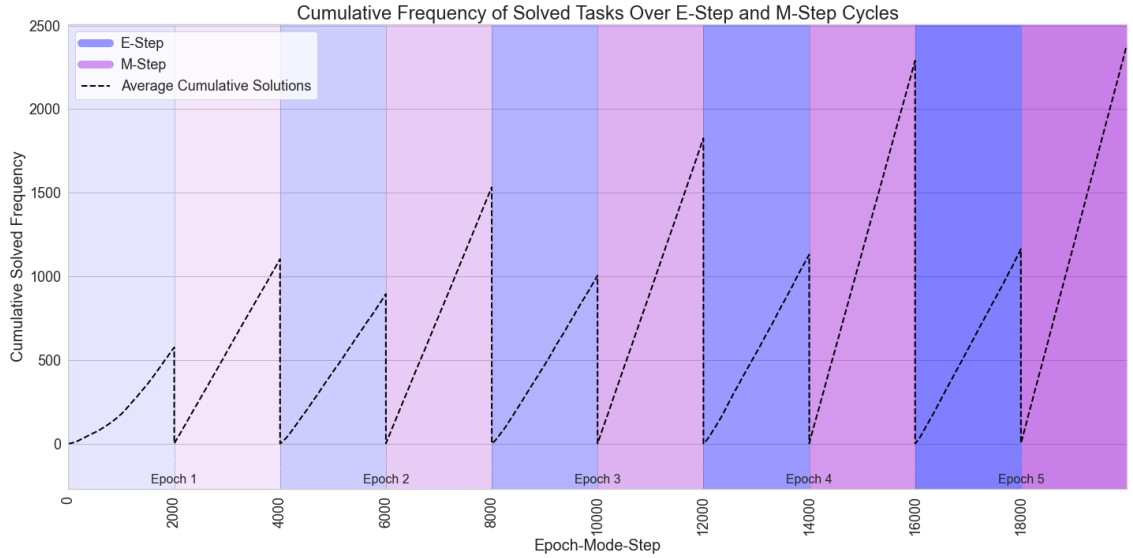

Figure 7: The plot displays the cumulative number of tasks solved (y-axis) against the number of steps (x-axis). Each step represents an iteration in the E-M cycle. The initial 2.000 steps correspond to the first E-step, marked with a blue background, followed by the first M-step spanning the next 2.000 steps (up to step 4.000), distinguished by a purple background. This pattern constitutes one complete epoch. The graph includes a dotted line representing the average number of tasks solved over all epochs, offering a benchmark for comparison. Furthermore, the intensity of the color hue in the plot encodes the temporal sequence of the epochs: brighter bars on the left signify earlier epochs (the first epoch), with the hue gradually darkening towards the right, culminating in the fifth and final epoch.

4 Discussion

In the following, first the experimental results are discussed, and the research questions addressed. Each step of FlowCoder's methodology is analyzed and compared, while including discussions of limitations and strategies for improvement. Next, we delve into more conceptual aspects of the model, discussing world models, the formation and representation of concepts, causality, the alignment of FlowCoder in a LOT, and finally its integration in a dual processing theory paradigm.

4.1 FlowCoder

Experiments Experiment 1 (subsection 3.1) suggests that FlowCoder is able to generalize to Out-of-Distribution tasks, given enough sleep. The model solves more tasks in inference than in training, including tasks of groups which had previously been unseen. Offline training, i.e. training on tasks from the generative model, and on remembered task-program pairs is indeed a crucial component of the model, facilitating generalization. This is especially apparent in comparison with experiment 2, in which the sleep phase was substantially reduced and the model performed worse on generalization. However, the sleep phase in experiment 1 was quite excessive and slowed the model down. It is therefore crucial to optimize the sleep phase and to find a balance between the gain of sleep and the cost of computation.

Experiment 2 (subsection 3.2) shows that the model has learnt the modes of some of the tasks to a degree in which it is able to solve them within one step during inference. As explained in [7], overfitting the model is not an issue, on the contrary, the model is only constrained by the amount of compute going into training. This suggests that when we invest in the model and train it to convergence, we can reap the benefits of the amortized sampler and attain a one-shot inference model.

The experiment shows that on average, the model resamples about 14% of the programs per task. This highlights an inefficiency of sampling, i.e. until the model learns the reward landscape and samples in proportion to it, it might resample unsuccessful programs. It is especially problematic in tasks that were not solved, meaning that the model is stuck, repeatedly proposing incorrect programs. Better exploration regimes could be employed to encourage a wider exploration of the program space in certain circumstances. FlowCoder's performance steadily increases with alternating E-M steps, suggesting that it is beneficial to train the generative model separately from the forward policy and having the two models bootstrap each other. The generative model proposes better representations of the program space while the forward policy selects better actions, proportional to the reward function, given the task at hand. This result confirms the findings of Hu *et al.* [38]. In experiment 2, 42 out of 95 tasks were solved during inference. As already mentioned, optimizing the sleep phase may improve the model's performance.

Additionally, other training regimes should be investigated. In my experiments, FlowCoder attempts to solve the random split of tasks chronologically, meaning the first task is seen with the initial weights of the models, while the last tasks are approached with already somewhat optimized weights. Instead, training on random minibatches of tasks could improve training. Moreover, curriculum learning, i.e. sorting the tasks by difficulty and starting with easier tasks before attempting more difficult ones, could be employed. Another question is for how long to train on a task or how often to switch tasks during training. The hyperparameters of the number of E- and M-steps,

number of epochs etc. were decided on empirically, mostly because the experiments were very time consuming, a more rigorous analysis on these details are an interesting topic for future work.

Minimum Description Length FlowCoder produces diverse solutions. Even in inference it proposes multiple programs solving the tasks, suggesting that it has found a multi-modal distribution. Being able to come up with multiple hypotheses of explaining observations is an important aspect of human cognition. Moreover, humans are able to hold multiple representations of states, exemplified by optical illusions such as the Necker Cube or the Spinning Dancer [7]. As expected, FlowCoder did not break syntactic symmetries, as shown in Table 4. Many, albeit inefficient, alternate solutions to a task were found. In principle, it is of course useful to avoid multiplying by 1, adding 0, and so on. Breaking syntactic symmetries can be done by favoring programs of minimum description length. DreamCoder does this by including an abstraction phase in which programs are refactored by finding commonalities between similar subroutines [24]. The authors employ E-graph matching and version spaces — data structures adept at compactly representing alternative iterations of a program and facilitating efficient set operations on these representations, respectively, to manage the combinatorial surge encountered during refactoring. Essentially, the system identifies sub-routines that have proven beneficial across numerous existing and newly discovered programs during the wake phase. It then abstracts these into novel concepts for future tasks, thereby refining and enhancing the DSL. Cerna & Kutsia [14] discuss different approaches for anti-unification, a method for finding generalizable structures between parse-trees. Implementing abstraction, was unfortunately beyond the scope of this thesis.

Scaling The main caveat of my method is that it is slow. Running the experiments took approximately a week and three days, respectively. It is somewhat difficult to compare the time efficiency of my method with other approaches for various reasons. Both DeepSynth and DreamCoder predict weights for the PCFG and then parallelize search processes on multiple CPUs. Ellis *et al.* [24] train DreamCoder for about a day on 20-100 CPUs and it takes around 10 wake-sleep cycles to converge in the list domain, meaning all tasks are solved. Instead, FlowCoder runs on a single GPU (with possible interruptions from other users on the cluster). DreamCoder shows that the refactoring algorithm used in abstraction is crucial in the list processing tasks, which was not implemented in FlowCoder. The authors of DeepSynth do not include the model query time in their computations and only investigate the performance of different search and sampling strategies. DeepSynth's HeapSearch algorithm solves all tasks, however it creates up to 1.000.000 programs per task, whereas FlowCoder only creates 80.000. So rather than trying to compare the different algorithms directly, FlowCoder's overall time complexity will be analyzed and whether FlowCoder scales will be discussed, because this is a paramount criterion for a model of cognition. Van Rooij's [81] P-Cognition thesis states that cognitive capacities are constrained by tractability - they are limited to functions that can be computed in polynomial time. The overall strategy I'm employing is to first compile a CFG, sample rules from it and sequentially form a trajectory, reconstruct the trajectory to AST format, and lastly evaluate the constructed program. In the following sections, I will analyze these parts in detail, suggest improvements and compare them to other methods.

[7] `https://en.wikipedia.org/wiki/Necker_cube` and `https://en.wikipedia.org/wiki/Spinning_dancer`, respectively

Rules and Type Inference When using a CFG, rule look-up can be done in $\mathcal{O}(1)$. Using a CFG is a nice workaround for my use case but of course human cognition does not come equipped with a CFG. Instead, when modeling with a typed λ-calculus, types must be inferred during the AST construction. Type inference in λ-calculus can be done in polynomial time [55]. Whether there is a cognitive analogue for a type system remains a point of discussion [71]. Goyal & Bengio [28] show how Transformer's attention can be seen as variable binding, thus implicitly inferring types. This was not investigated in this research because of time constraints but is an interesting topic for future research.

Program Sampling, Representation, and Embedding As outlined in section 2.2(Sampling Programs), FlowCoder constructs programs by sampling rules sequentially. The bulk of the computation is done by the Transformer, specifically the self-attention mechanism, with a time complexity of $\mathcal{O}(n^2)$ per forward pass [43]. The auto-regressive trajectory construction therefore has a time complexity of $\mathcal{O}(n^3)$. However, ASTs can be constructed in various orders. Ideally, additionally to predicting the value of the next node, we could let the GFlowNet predict which node to expand next. However, this would have meant creating an additional model as a node predictor, which increases FlowCoder's complexity. As mentioned in section 2.2, graph neural networks could be useful for this purpose. Currently, each rule of the CFG is embedded, alternatively additional information like parent, child, and argument index could be parameterized explicitly as in DreamCoder. Moreover, type information, depth, or other variables could be encoded in the node representation. Allamanis *et al.* [2] use Gated Graph Neural Networks to represent both syntactic and semantic information by including data flow and type hierarchy signals. The authors formalize how to turn ASTs to graphs. Wang *et al.* [84] propose a semantic program embedding, learnt from program execution traces. Ibarz *et al.* [39] present a generalist neural algorithmic learner by leveraging GNNs. Zhang *et al.* [90] propose a novel neural AST representation of source code. Oliveira & Löh [63] develop abstract syntax graphs (rather than trees) to preserve sharing and recursion within a DSL. After sampling a trajectory, the program is reconstructed to AST format which can be done in $\mathcal{O}(n)$. GNNs may be a more natural way to represent tree-structured data such as ASTs and would eliminate the need to format back and forth between linearized and tree representations of AST. However, Joshi [40] shows that Transformers are actually a special case of GNNs, namely they are fully connected graphs. Therefore, I decided to simply use the Transformer architecture as is.

Zhang *et al.* [89] demonstrate evidence for a hyperbolic geometry in the neural representation of space in the Hippocampus. Given that we are working with trees, a hyperbolic space rather than Euclidean, might be beneficial. Cetin *et al.* [15] show that agents with a hyperbolic representation of latent state space substantially outperform their Euclidean counterparts in deep reinforcement learning benchmark tasks. Auyespek *et al.* [4] confirm that tree-structured data is better represented in hyperbolic space than in Euclidean space. Similar results are established by Khan *et al.* [44] and Nickel & Nickel & Kiela [61]. Lu *et al.* [54] propose a hyperbolic function embedding method and show that it is more efficient in terms of computation as well as storage compared to other graph embedding methods. Nevertheless, after an initial investigation of hyperbolic AST embeddings, I decided not to pursue this for two reasons. First, because the trees in the experiments are very shallow (up to depth 4), so the nodes are unlikely to overlap in latent space, and second, because I was constrained by time. However, hyperbolic embedding spaces show promise

and should be investigated in the future.

Evaluation The time complexity of evaluating an AST is linear ($\mathcal{O}(n)$) with respect to the number of nodes, assuming constant-time operations, such as a simple arithmetic operation or a variable lookup, at each node ($\mathcal{O}(1)$). If the operation at each node involves more complex processes, such as function calls, recursive evaluations, or complex computations, the time complexity can increase. For instance, if a node operation has a complexity of $\mathcal{O}(m)$, then the overall complexity becomes $\mathcal{O}(n \times m)$. The structure of the AST can influence the complexity. For instance, in a highly unbalanced tree, certain types of traversals may become less efficient. Again, different AST representations would therefore be useful to investigate. Since we are working in an executable language, rather than e.g. natural language, the sampling order and AST representation may affect the performance of the model. Instead of first constructing the tree and then evaluating it, we could predict terminal nodes first in a bottom-up fashion and evaluate the tree (or subtrees) with successive predictions as is done e.g. in the heap-search algorithm [25]. Other types of information flow could also propagate through the tree, enriching the node embeddings, as previously discussed.

Levenshtein as Reward The Levenshtein edit distance algorithm (Appendix E), when implemented using dynamic programming, can be computed with a time complexity of $\mathcal{O}(m \times n)$, where m and n are the lengths of the two input strings. Andoni *et al.* [3] present a near-linear time approximation of the edit distance within a polylogarithmic factor.
However, the Levenshtein edit distance is not optimal either. It captures the syntactic difference between the outputs rather than the semantic nuances. Consider the task `[1,2,3,4]` $\rightarrow$ `[1,4,9,16]`. The correct program squares each element in the input list. Say the model constructs two programs. Program f cubes the input list, which outputs: `[1,8,27,64]`. Program g takes the last element of the list and moves it to index 2: `[1,4,2,3]`. Semantically, f is closer to the correct program since it knows it must map each element in the list to the power of a variable, but it used the wrong variable, whereas g is not similar to the correct program at all. However, the Levenshtein distance would be higher (and thus the reward lower) for the output of f. The Mean Squared Error, Hamming distance, and similar approaches were explored but deemed unfitting and less informative than the Levenshtein distance. Interpreting the output lists as vectors and calculating e.g. a cosine similarity was also considered, however vectors of varying lengths cannot be compared directly and padding the lists would skew the outcome [8].

4.2 World Model

In the programming paradigm, the model learns the structure of the world by adhering to the syntactic relations $\mathcal{S}$ of the DSL $\mathcal{D}$ and applying type inference or, as in DeepSynth and FlowCoder, directly from the CFG. The combination of the CFG (or alternatively type inference) and the reward R can be seen as an *explicit* world model. The Transformer can be seen as a generative model $p_\theta(x, s)$ that learns to map $T(x, s) = z \propto R$, thereby learning an *implicit* world model. The forward policy π can be regarded as an amortized approximation of R, a DAG over the state space, and represents a marginal distribution over trajectories leading to programs ρ. In FlowCoder, R is a static heuristic, fairly sufficient in the list editing domain. However, we may ask how it should be represented in a more complex and dynamic environment. The "Reward-is-Enough" hypothesis

[8] See Han *et al.* [32] for a summary of the mentioned approaches

by Silver *et al.* [75] suggests that an agent maximizing a reward while acting on its environment may be sufficient to elicit intelligence and the abilities that are associated with it. Colas *et al.* [18] argue, that humans develop by intrinsic motivation, i.e. they set their own goals and rewards. The authors define how an agent can set a goal construct which pairs a compact goal representation with a function that measures its achievement. Reward has also been hypothesized to originate in the environment and be a consequence of computing the distance to intrinsic goals [41]. Rule *et al.* [71] use "hacking" as a metaphor for learning in cognitive development and highlight the importance of intrinsic motivation. A child does not only solve external goals but instead plays, finds sub-goals, and maintains a diverse network of goals. The reward R could in principle reflect a range of values, such as aesthetics, accuracy, usefulness, etc. which is a thrilling direction for future research.

Bayesian Models Griffiths *et al.* [29] advocate for a hierarchical Bayesian model as a model for cognition. FlowCoder can be extended to reflect a Bayesian posterior by parameterizing $R_\theta(z|x) = p_\theta(x,z) = p_\theta(z)p_\theta(x|z)$, such that the policy π samples in proportion to the posterior distribution. The distribution can be made explicit by using a distributed probabilistic CFG - a neuralPCFG as introduced in Kim *et al.* [46] and adapted in GFlowNet-EM [38]. However, this necessitates computing the marginal likelihood which is challenging. Of course, it can again be approximated by using GFlowNets or other methods. DreamCoder marginalizes over a beam of programs for each task. Markov Chain Monte Carlo (MCMC) samples from the posterior distribution without computing the marginal likelihood directly. However, they can be slow to converge and have difficulty with mixing modes. Variational Inference (VI) approximates the posterior with a simpler distribution, which is computationally more tractable. It transforms the problem into optimization. See Appendix G for a brief comparison between VI and GFlowNets. Gulwani *et al.* [31] give a detailed survey and summarization of program synthesis and commonly used methods.

Energy-Based Models Alternatively, as briefly discussed in section 2.2 (Reward), the world model could also be represented by an Energy-Based Model. I will briefly introduce the idea of EBMs in the context of our use case and outline the procedure for their application.

The energy function $E_\theta(x)$ in EBMs, as defined by LeCun et al. [51], is a real-valued function where x represents the data and θ the model parameters. EBMs are designed such that more probable (or desirable) data points are assigned lower energy values, reflecting their likelihood of being from the real distribution. Conversely, less probable data points receive higher energy values, indicating divergence from the expected distribution.
The output of the energy function $E_\theta(x)$ is unnormalized. The probability of data point x under this model is given by the Boltzmann distribution:

$$P_\theta(x) = \frac{e^{-E_\theta(x)}}{Z_\theta} \tag{8}$$

where Z_θ is the partition function. Computing Z_θ, which is the sum of $e^{-E_\theta(x)}$ over all possible x, is computationally challenging. Contrastive Divergence (CD), introduced by Hinton [35], provides a practical method to approximate this function through a two-phase process.
In our specific case, a task x is represented by an input-output pair $(x_{\text{in}}, x_{\text{out}})$. In the *positive phase* of CD, we compute the energy of the real task $E_\theta(x)$, which serves as a baseline measure

of task feasibility. The *negative phase* involves sampling a program ρ using the policy $\pi_\phi(\rho|x)$, executing ρ with x_{in} to produce a predicted output $\tilde{x}_{\text{out}}$ (combined as $\tilde{x}$), and then computing the energy for this modified task output $E_\theta(\tilde{x})$. The discrepancy in energy between the real and predicted outputs is leveraged to refine the EBM:

$$\Delta\theta = \eta(\nabla E_\theta(x) - \nabla E_\theta(\tilde{x})) \tag{9}$$

where η is the learning rate. This training process enables the EBM to distinguish between the expected and predicted outputs, thus providing a detailed reward function that enhances the policy's approximation of the posterior distribution. A significant advantage of using EBMs in this context is their domain-agnostic nature, allowing flexibility in application across various task domains, extending beyond list editing. The reward function could be optimized further, incorporating evaluations of partial programs, tree-shape, length (e.g. to optimize for MDL), or other features.

The benefit of Energy-Based Model over than Bayesian models is that EBMs are implicit, meaning we can make less assumptions about the model which makes this method more flexible.

Biological Plausibility In one way or another, probabilistic world models are useful and necessary. Pouget *et al.* [69] present strong evidence that the human brain can represent probabilistic distributions and show how these may be represented in neural circuits. Moreover, they suggest that humans perform probabilistic inference.
The predictive processing paradigm and many other associated theories of cognition posit that instead of viewing the brain as a passive data collector, the brain is a query mechanism, primarily engaged in top-down prediction [11, 13, 60]. The primary function becomes producing predictions about imminent sensory inputs. What ascends is the prediction error, the mismatch between predictions and actual sensory input.

We also know that movement is organized hierarchically, initiated by the Cortex, after which the Cerebellum compares the intended movement with the actual movement and sends an error message to the Cortex to improve the accuracy of the subsequent movement [48]. This hints at the Cerebellum's function of representing some kind of prior and being able to compare what is with what ought to be. Recent research suggests that the Cerebellum, containing around 50 % of the brains neurons, may be responsible for more than just motor-action and could represent a world model [76–78]. More research is needed however, to better understand the specialized brain functions and representations.

As I have shown in this research, the line between probabilistic models such as explicit PCFGs and using neural networks, such as the Transformer, to represent an implicit world model is blurred. Griffiths *et al.* [30] propose to refrain from seeing Bayesian models and neural networks as opposing but rather as complimentary approaches to understand cognition, distinguished by Marr & Poggio's [57] 3 levels of analysis, where Bayesian models correspond to the computational level and neural networks to algorithmic and implementational levels of analysis.

4.3 Concepts

Both discrete symbolic and continuous sub-symbolic representations may be necessary to understand human cognition. Symbols limit bandwidth, are robust to noise, and enable re-use for

systematic generalization [21, 53]. Humans do not start out from a blank slate [49, 68] but rather have some innate abilities for intuitive physics and psychology [79, 80], and to recognize symmetry and other relations [21]. Symbols must be grounded and abstracted bottom-up from sensorimotor interaction with an environment [85]. d'Avila Garcez & Lamb [19] outline how once neural networks are trained, symbols can be extracted. Consider a neural network that learns certain regularities and recognizes that whenever it sees features of type A, features of type B are also present, but features of type C are not. This implicit rule can be made explicit by converting it into symbols: $\forall x A(x) \rightarrow (\exists y B(y) \wedge \neg \exists z C(z))$. Piantadosi [67] demonstrates how to construct a Language of Thought through combinatorial-logic-like language, which is as expressive as λ-calculus but does not necessitate primitives. This is done via Church Encoding, where primitives are exchanged by higher-order functions. Piantadosi [67] asserts that symbols have no inherent meaning, instead, meaning emerges from the dynamic relations between symbols. I.e., only the conceptual role and the dynamics of the symbols define meaning. This perspective aligns with Conceptual Role Semantics, which posits that meaning and content are derived from the roles symbols play in thought and language. Symbols are static, but their distributed representations as compositions of embeddings afford a rich internal structure which may be interpreted as semantics. Therefore, embedding programs may be a crucial aspect to go beyond syntax and learn a semantic grammar, which is an improvement of FlowCoder over DreamCoder. Do & Hasselmo [23] explore how the brain can manipulate symbols. They suggest that sequences and neural populations can provide representations that match the requirements for symbols, and that reinforcement learning and Bayesian inference can give the brain the capacity for goal-directed planning. Moreover, they suggest that neural operations can be mapped to λ-calculus, bridging the gap between connectionism and classical AI and thereby help us understand the circuit mechanism behind manipulating symbols.

The execution of functions within a computational LOT may be necessary for the generation of new knowledge and the expansion of the DSL. This process involves the construction of complex concepts from basic primitives, akin to the construction of mathematical knowledge from axioms. In other words, every new concept is the result of a computational process, thereby forcing its correctness. Consequently, since every concept is a trajectory, aligning with constructivist principles, when it is integrated into a world model, coherence is assured.

Causal Models Humans are endowed with an intuitive grasp of causality, which functions as the underpinning for both predictive and counterfactual reasoning [49]. This enables humans to extrapolate beyond the confines of empirical data, to conjure hypothetical worlds e.g. in imagination, play, and dreaming. A causal model typically consists of a set of variables and a set of directed edges between these variables, often represented as a DAG. The vertices in this graph denote variables, which, in this context, could be seen as cognitive states. The directed edges signify causal relationships, pointing from cause to effect. Causal models facilitate interventions, counterfactuals, and causal inference. Intervention is the ability to model the outcome of purposeful changes, represented mathematically as "do" operations [64]. For example, if one were to intervene to set $X = x$, the model would enable us to compute the distribution over Y post-intervention. Counterfactual reasoning allows us to traverse back in time, in a sense, and assess alternative realities — what would have happened to Y if X had been different? Kıcıman *et al.* [47] show that LLMs (particularly GPT-4) are able to *mimic* causality but do not show evidence of real

causality. Zečević *et al.* [88] are skeptical of LLM's causal abilities and show that even when they perform well on causal tasks, they are more likely to learn correlations of causal facts. Programs are inherently causal and support counterfactual reasoning [16]. Using the Transformer, being able to represent a rich representation and forcing it to create executable language - programs, may be an efficient strategy that supports causality. For instance, FlowCoder can use imagined data to make inferences about resulting programs. However, incorporating causality in probabilistic models and neural networks is still an ongoing and somewhat recent field of research and further developments are necessary to compare and contrast how the brain implements it and how we can build models that are truly causal [73].

4.4 System 1 & System 2

Kahneman [42] describes two distinct systems that characterize the dual aspects of human cognition:
System 1 operates automatically and quickly, with little or no effort and no sense of voluntary control. It encompasses intuitive judgments and perceptual associations — it allows for rapid, heuristic-based processing that is often subconscious. System 1 uses the world model, providing immediate perceptual hypotheses and predictions that guide behavior in a dynamic manner without the need for conscious deliberation.

System 2 allocates attention to the effortful mental activities that demand it, including complex computations. It is associated with the conscious, rational mind and is deliberate, effortful, and orderly. System 2 aligns with the process of the inferential and constructive process of building a trajectory, leading to a program likely to match a given specification. Adjusting the forward policy is akin to the effortful correction or modulation of System 1's rapid predictions when errors are detected or when more complex reasoning is required.

For example, when learning to play the piano, initial efforts require conscious attention and deliberate practice to master certain phrases (System 2). Over time, as these phrases become internalized, they can be executed more automatically and intuitively (System 1).

Another way of thinking about this by distinguishing *habitual* vs. *controlled* processing where the former is *implicit* and the latter *explicit* [28]. System 1 is often conceptualized as *procedural*, *parallel*, *continuous*, and *subsymbolic*, whereas System 2 is seen as *declarative*, *sequential*, *discrete*, and *symbolic*. There is a growing literature on discussing the necessity and suggesting methods of incorporating System 1 and System 2 [12, 50, 53, 59, 62]. FlowCoder attempts to take a step in this direction, bridging the gap between the two systems, by learning an implicit world model and learning a forward policy that constructs programs. In a future work, FlowCoder should learn to refactor and abstract programs, and grow its Domain Specific Language.

4.5 Limitations & Future Work

FlowCoder is far from perfect. Many tasks were unsolved and only a very shallow tree depth was permitted. I will briefly summarize possible optimizations. FlowCoder could benefit from better AST and DSL representations (see. e.g. [63, 90]), hyperbolic embedding spaces that are better able to capture tree structures [54], and partial program evaluations, e.g. as is done in the HeapSearch algorithm [25]. By including a backward policy, we can relax the order of the sampling strategy

and let FlowCoder predict how to expand an AST. Moreover, in this research, many important constraints have been relaxed, such as breaking syntactic symmetries, optimizing for Minimum Description Length, and growing the DSL, all of which can be achieved by incorporating abstraction, as is done in [24]. FlowCoder's training can be further optimized by improving the balance between wake and sleep phases and improved exploration. In order to make FlowCoder applicable in other, more complex domains, the reward needs to be improved, e.g. by parameterizing it as an EBM. Moreover, the model should be tested in domains other than list-editing, incorporating polymorphic types.

5 Conclusion

FlowCoder represents a step forward in the field of program synthesis and neurosymbolic AI, attempting to bridge the gap between the flexibility of neural networks and the precision of symbolic reasoning. The use of a Transformer architecture combined with GFlowNet within the FlowCoder framework enables efficient one-shot inference and extrapolation to novel Out-of-Distribution tasks. Moreover, the integration of replay and fantasy techniques enhances the model's learning process, allowing it to retain and refine solutions over time, further improving its performance and generalization abilities. FlowCoder's design, leveraging both explicit and implicit world models, provides a promising approach to program synthesis. However, optimization techniques and extensions to other domains are required. The experiments revealed a trade-off between generalization capabilities and computational costs, particularly during the sleep phase. Future work could focus on optimizing this aspect to achieve a better balance, enhancing the model's efficiency without sacrificing its learning and generalization abilities. The study touches upon the concept of intrinsic motivation and self-generated goals, which could be a fertile ground for further exploration. The discussion highlights the ongoing debate between symbolic and subsymbolic models, touching upon dual-process theories of cognition. Further theoretical research is needed to deepen our understanding of these models' relationship and how they can be effectively combined to create AI that more closely aligns with human cognitive processes.

Acronyms

AI Artificial Intelligence. 1, 6, 32

AST Abstract Syntax Tree. 14, 15, 25–27, 31, 32

CFG Context Free Grammar. 9, 10, 14–16, 25–28

CRS Conceptual Role Semantics. 30

DAG Directed Acyclic Graph. 12, 27, 30

DC DreamCoder. 8, 9, 11

DFS Depth First Search. 10

DSL Domain Specific Language. 3, 7–11, 25, 27, 30–32, 41

EBM Energy-Based Model. 17, 28, 29, 32

EM Expectation-Maximization. 13

GFlowNet Generative Flow Network. 1, 10–12, 28

GNN Graph Neural Network. 15, 26

GOFAI Good Old-Fashioned AI. 6

IO Input-Output. 15

LLM Large Language Model. 1, 30, 31

LOT Language of Thought. 1, 6, 7, 11, 24, 30

MCMC Markov Chain Monte Carlo. 28

MDL Minimum Description Length. 6, 8, 11, 32

MLP Multi-Layer Perceptron. 16

OOD Out-of-Distribution. 1, 6, 11, 24, 32

PCFG Probabilistic Context Free Grammar. 9, 10, 25, 29

SOTA State-of-the-Art. 11

TB Trajectory Balance. 12

VI Variational Inference. 28

Bibliography

1. Al Roumi, F., Marti, S., Wang, L., Amalric, M. & Dehaene, S. Mental Compression of Spatial Sequences in Human Working Memory Using Numerical and Geometrical Primitives. *Neuron* **109,** 2627–2639.e4. ISSN: 0896-6273. (2022) (Aug. 2021) (cit. on p. 6).
2. Allamanis, M., Khademi, M. & Brockschmidt, M. Learning to Represent Programs with Graphs (2018) (cit. on pp. 15, 26).
3. Andoni, A., Krauthgamer, R. & Onak, K. *Polylogarithmic Approximation for Edit Distance and the Asymmetric Query Complexity* in *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA* (IEEE Computer Society, 2010), 377–386 (cit. on p. 27).
4. Auyespek, T., Mach, T. & Assylbekov, Z. Hyperbolic Embedding for Finding Syntax in BERT (cit. on p. 26).
5. Bengio, E., Jain, M., Korablyov, M., Precup, D. & Bengio, Y. *Flow Network Based Generative Models for Non-Iterative Diverse Candidate Generation* in *Advances in Neural Information Processing Systems* **34** (Curran Associates, Inc., 2021), 27381–27394. (2022) (cit. on pp. 10, 12).
6. Bengio, Y., Lecun, Y. & Hinton, G. Deep Learning for AI. *Communications of the ACM* **64,** 58–65 (2021) (cit. on p. 6).
7. Bengio, Y. *et al.* GFlowNet Foundations. *Journal of Machine Learning Research* **24,** 1–55. ISSN: 1533-7928. (2023) (2023) (cit. on pp. 17, 24).
8. Berwick, R. C., Pietroski, P., Yankama, B. & Chomsky, N. Poverty of the Stimulus Revisited. *Cognitive Science* **35,** 1207–1242. ISSN: 1551-6709. (2024) (2011) (cit. on p. 6).
9. Bieber, D., Sutton, C., Larochelle, H. & Tarlow, D. *Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks* Oct. 2020. (2023) (cit. on p. 15).
10. Boicho, D. G. K. J. H. & Kokinov, N. *The Analogical Mind: Perspectives from Cognitive Science* (MIT press, 2001) (cit. on p. 6).
11. Bottemanne, H., Longuet, Y. & Gauld, C. [The predictive mind: An introduction to Bayesian Brain Theory]. *L'Encephale* **48,** 436–444. ISSN: 0013-7006 (Aug. 2022) (cit. on p. 29).
12. Cartuyvels, R., Spinks, G. & Moens, M.-F. Discrete and Continuous Representations and Processing in Deep Learning: Looking Forward. *AI Open* **2,** 143–159. ISSN: 26666510. arXiv: 2201.01233 [cs]. (2023) (2021) (cit. on p. 31).

13. Caucheteux, C., Gramfort, A. & King, J.-R. Evidence of a Predictive Coding Hierarchy in the Human Brain Listening to Speech. *Nature Human Behaviour* **7,** 430–441. ISSN: 2397-3374. (2023) (Mar. 2023) (cit. on p. 29).

14. Cerna, D. M. & Kutsia, T. *Anti-Unification and Generalization: A Survey* Feb. 2023. arXiv: 2302.00277 [cs]. (2023) (cit. on p. 25).

15. Cetin, E., Chamberlain, B., Bronstein, M. & Hunt, J. J. *Hyperbolic Deep Reinforcement Learning* Oct. 2022. arXiv: 2210.01542 [cs]. (2023) (cit. on p. 26).

16. Chater, N. & Oaksford, M. Programs as Causal Models: Speculations on Mental Programs and Mental Representation. *Cognitive Science* **37,** 1171–1191. ISSN: 1551-6709. (2024) (2013) (cit. on p. 31).

17. Chollet, F. *On the Measure of Intelligence* Nov. 2019. arXiv: 1911.01547 [cs]. (2024) (cit. on p. 6).

18. Colas, C., Karch, T., Sigaud, O. & Oudeyer, P.-Y. *Autotelic Agents with Intrinsically Motivated Goal-Conditioned Reinforcement Learning: A Short Survey* July 2022. arXiv: 2012.09830 [cs]. (2023) (cit. on p. 28).

19. D'Avila Garcez, A. & Lamb, L. C. Neurosymbolic AI: The 3rd Wave. *arXiv:2012.05876 [cs].* arXiv: 2012.05876 [cs]. (2022) (Dec. 2020) (cit. on p. 30).

20. de Bruijn, N. G. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)* **75,** 381–392. ISSN: 1385-7258. (2023) (Jan. 1972) (cit. on p. 14).

21. Dehaene, S., Al Roumi, F., Lakretz, Y., Planton, S. & Sablé-Meyer, M. Symbols and Mental Programs: A Hypothesis about Human Singularity. *Trends in Cognitive Sciences* **26,** 751–766. ISSN: 13646613. (2022) (Sept. 2022) (cit. on pp. 6, 7, 30).

22. Deleu, T. *et al. Bayesian Structure Learning with Generative Flow Networks* June 2022. arXiv: 2202.13903 [cs, stat]. (2024) (cit. on p. 10).

23. Do, Q. & Hasselmo, M. E. Neural Circuits and Symbolic Processing. *Neurobiology of Learning and Memory* **186,** 107552. ISSN: 1074-7427. (2023) (Dec. 2021) (cit. on p. 30).

24. Ellis, K. *et al. 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* (ACM, Virtual Canada, June 2021), 835–850. ISBN: 978-1-4503-8391-2. (2022) (cit. on pp. 8, 9, 13, 25, 32).

25. Fijalkow, N. *et al. Scaling Neural Program Synthesis with Distribution-based Search* Oct. 2021. arXiv: 2110.12485 [cs]. (2023) (cit. on pp. 9, 10, 17, 27, 31, 41).

26. Fodor, J. A. & Pylyshyn, Z. W. Connectionism and Cognitive Architecture: A Critical Analysis. *Cognition* **28,** 3–71. ISSN: 0010-0277 (Mar. 1988) (cit. on p. 6).

27. Friston, K. *et al.* World Model Learning and Inference. *Neural Networks* **144,** 573–590. ISSN: 0893-6080. (2023) (Dec. 2021) (cit. on p. 6).

28. Goyal, A. & Bengio, Y. Inductive Biases for Deep Learning of Higher-Level Cognition. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* **478,** 20210068. (2023) (Oct. 2022) (cit. on pp. 26, 31).

29. Griffiths, T. L., Kemp, C. & Tenenbaum, J. B. Bayesian Models of Cognition. *BAYESIAN MODELS* (cit. on p. 28).

30. Griffiths, T. L., Zhu, J.-Q., Grant, E. & McCoy, R. T. Bayes in the age of intelligent machines. arXiv:2311.10206 [cs]. `http://arxiv.org/abs/2311.10206` (Nov. 2023) (cit. on p. 29).

31. Gulwani, S., Polozov, O. & Singh, R. *Program Synthesis Foundations and Trends in Programming Languages* **4.2017, 1-2.** ISBN: 978-1-68083-292-1 (Now Publishers, Hanover, MA Delft, 2017) (cit. on p. 28).

32. Han, J., Pei, J. & Tong, H. *Data Mining: Concepts and Techniques* (Morgan kaufmann, 2022) (cit. on pp. 13, 27).

33. He, Q., Sedoc, J. & Rodu, J. *Trees in Transformers: A Theoretical Analysis of the Transformer's Ability to Represent Trees* Dec. 2021. arXiv: `2112.11913 [cs]`. (2023) (cit. on p. 15).

34. Hinton, G. *How to Represent Part-Whole Hierarchies in a Neural Network* Feb. 2021. arXiv: `2102.12627 [cs]`. (2023) (cit. on p. 6).

35. Hinton, G. E. Training Products of Experts by Minimizing Contrastive Divergence. *Neural computation* **14,** 1771–1800 (2002) (cit. on p. 28).

36. Hinton, G. E., Dayan, P., Frey, B. J. & Neal, R. M. The" Wake-Sleep" Algorithm for Unsupervised Neural Networks. *Science (New York, N.Y.)* **268,** 1158–1161 (1995) (cit. on pp. 8, 13).

37. Hofstadter, D. & Sander, E. *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking* (Basic Books, 2013) (cit. on p. 6).

38. Hu, E. J. *et al. GFlowNet-EM for Learning Compositional Latent Variable Models* en. in *Proceedings of the 40th International Conference on Machine Learning* (PMLR, July 2023), 13528–13549 (cit. on pp. 10, 13, 14, 24, 28).

39. Ibarz, B. *et al. A Generalist Neural Algorithmic Learner* Dec. 2022. (2022) (cit. on pp. 15, 26).

40. Joshi, C. Transformers Are Graph Neural Networks. *The Gradient* **7** (2020) (cit. on p. 26).

41. Juechems, K. & Summerfield, C. Where Does Value Come From? en. *Trends in Cognitive Sciences* **23,** 836–850. ISSN: 13646613 (Oct. 2019) (cit. on p. 28).

42. Kahneman, D. *Thinking, Fast and Slow* ISBN: 978-0-374-27563-1 (Farrar, Straus and Giroux, New York, 2011) (cit. on p. 31).

43. Keles, F. D., Wijewardena, P. M. & Hegde, C. *On the Computational Complexity of Self-Attention* 2022. arXiv: `2209.04881 [cs.LG]` (cit. on p. 26).

44. Khan, R., Nguyen, T. V. & Srinivasan, S. H. Hyperbolic Representations of Source Code (cit. on p. 26).

45. Khan, S. *et al.* Transformers in Vision: A Survey. *ACM Computing Surveys* **54,** 200:1–200:41. ISSN: 0360-0300. (2024) (Sept. 2022) (cit. on p. 10).

46. Kim, Y., Dyer, C. & Rush, A. *Compound Probabilistic Context-Free Grammars for Grammar Induction* in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (Association for Computational Linguistics, Florence, Italy, July 2019), 2369–2385. (2023) (cit. on pp. 10, 13, 28).

47. Kıcıman, E., Ness, R., Sharma, A. & Tan, C. *Causal Reasoning and Large Language Models: Opening a New Frontier for Causality* May 2023. arXiv: 2305.00050 [cs, stat]. (2023) (cit. on p. 30).

48. Kolb, B., Whishaw, I. Q., Teskey, G. C., Whishaw, I. Q. & Teskey, G. C. *An Introduction to Brain and Behavior* (Worth New York, 2001) (cit. on p. 29).

49. Lake, B. M., Ullman, T. D., Tenenbaum, J. B. & Gershman, S. J. Building Machines That Learn and Think like People. *Behavioral and Brain Sciences* **40,** e253. ISSN: 0140-525X, 1469-1825. (2023) (2017) (cit. on pp. 7, 30).

50. LeCun, Y. A path towards autonomous machine intelligence version 0.9. 2, 2022-06-27. *Open Review* **62** (2022) (cit. on pp. 6, 31).

51. LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M. & Huang, F. J. A Tutorial on Energy-Based Learning (cit. on p. 28).

52. Levenshtein, V. I. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* **10,** 707 (Feb. 1966) (cit. on p. 17).

53. Liu, D. *et al. Discrete-Valued Neural Communication* in *Advances in Neural Information Processing Systems* **34** (Curran Associates, Inc., 2021), 2109–2121. (2022) (cit. on pp. 30, 31).

54. Lu, M. *et al.* Hyperbolic Function Embedding: Learning Hierarchical Representation for Functions of Source Code in Hyperbolic Space. *Symmetry* **11,** 254. ISSN: 2073-8994. (2023) (Feb. 2019) (cit. on pp. 26, 31).

55. Mairson, H. G. Linear Lambda Calculus and PTIME-completeness. *Journal of Functional Programming* (cit. on p. 26).

56. Malkin, N., Jain, M., Bengio, E., Sun, C. & Bengio, Y. *Trajectory Balance: Improved Credit Assignment in GFlowNets* Oct. 2022. arXiv: 2201.13259 [cs, stat]. (2022) (cit. on p. 12).

57. Marr, D. & Poggio, T. From understanding computation to understanding neural circuitry (1976) (cit. on p. 29).

58. Martins, M. D., Laaha, S., Freiberger, E. M., Choi, S. & Fitch, W. T. How Children Perceive Fractals: Hierarchical Self-Similarity and Cognitive Development. *Cognition* **133,** 10–24. ISSN: 0010-0277. (2024) (Oct. 2014) (cit. on p. 6).

59. Matsuo, Y. *et al.* Deep Learning, Reinforcement Learning, and World Models. *Neural Networks* **152,** 267–275. ISSN: 0893-6080. (2024) (Aug. 2022) (cit. on p. 31).

60. Mazzaglia, P., Verbelen, T., Çatal, O. & Dhoedt, B. The Free Energy Principle for Perception and Action: A Deep Learning Perspective. *Entropy* **24,** 301. ISSN: 1099-4300. arXiv: 2207.06415 [cs, q-bio]. (2023) (Feb. 2022) (cit. on p. 29).

61. Nickel, M. & Kiela, D. *Poincaré Embeddings for Learning Hierarchical Representations* in *Advances in Neural Information Processing Systems* **30** (Curran Associates, Inc., 2017). (2023) (cit. on p. 26).

62. Nye, M., Tessler, M. H., Tenenbaum, J. B. & Lake, B. M. *Improving Coherence and Consistency in Neural Sequence Models with Dual-System, Neuro-Symbolic Reasoning* Dec. 2021. arXiv: 2107.02794 [cs]. (2023) (cit. on p. 31).

63. Oliveira, B. C. D. S. & Löh, A. *Abstract Syntax Graphs for Domain Specific Languages* in *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation* (ACM, Rome Italy, Jan. 2013), 87–96. ISBN: 978-1-4503-1842-6. (2023) (cit. on pp. 26, 31).

64. Pearl, J. *Theoretical Impediments to Machine Learning With Seven Sparks from the Causal Revolution* in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (ACM, Marina Del Rey CA USA, Feb. 2018), 3–3. ISBN: 978-1-4503-5581-0. (2023) (cit. on p. 30).

65. Pearson, J. The Human Imagination: The Cognitive Neuroscience of Visual Mental Imagery. *Nature Reviews Neuroscience* **20,** 624–634. ISSN: 1471-0048. (2024) (Oct. 2019) (cit. on p. 6).

66. Peng, H., Li, G., Zhao, Y. & Jin, Z. *Rethinking Positional Encoding in Tree Transformer for Code Representation* in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing* (Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, Dec. 2022), 3204–3214. (2023) (cit. on p. 15).

67. Piantadosi, S. T. The Computational Origin of Representation. *Minds and Machines* **31,** 1–58. ISSN: 0924-6495, 1572-8641. (2022) (Mar. 2021) (cit. on p. 30).

68. Pinker, S. *The Blank Slate: The Modern Denial of Human Nature* (Penguin, 2003) (cit. on p. 30).

69. Pouget, A., Beck, J. M., Ma, W. J. & Latham, P. E. Probabilistic brains: knowns and unknowns. eng. *Nature Neuroscience* **16,** 1170–1178. ISSN: 1546-1726 (Sept. 2013) (cit. on p. 29).

70. Rauss, K. & Pourtois, G. What Is Bottom-Up and What Is Top-Down in Predictive Coding? *Frontiers in Psychology* **4.** ISSN: 1664-1078. (2022) (2013) (cit. on p. 6).

71. Rule, J. S., Tenenbaum, J. B. & Piantadosi, S. T. The Child as Hacker. *Trends in Cognitive Sciences* **24,** 900–915. ISSN: 13646613. (2022) (Nov. 2020) (cit. on pp. 7, 26, 28).

72. Sablé-Meyer, M., Ellis, K., Tenenbaum, J. & Dehaene, S. A Language of Thought for the Mental Representation of Geometric Shapes. *Cognitive Psychology* **139,** 101527. ISSN: 00100285. (2022) (Dec. 2022) (cit. on p. 11).

73. Schölkopf, B. *et al.* Toward Causal Representation Learning. *Proceedings of the IEEE* **109,** 612–634. ISSN: 1558-2256. (2023) (May 2021) (cit. on p. 31).

74. Schwartz, O. & Giraldo, L. G. S. Behavioral and Neural Constraints on Hierarchical Representations. *Journal of Vision* **17,** 13. ISSN: 1534-7362. (2024) (Mar. 2017) (cit. on p. 6).

75. Silver, D., Singh, S., Precup, D. & Sutton, R. S. Reward Is Enough. *Artificial Intelligence* **299,** 103535. ISSN: 00043702. (2024) (Oct. 2021) (cit. on p. 28).

76. Silver, M. A. Cognitive Neuroscience: Functional Specialization in Human Cerebellum. *Current biology : CB* **28,** R1256–R1258. ISSN: 0960-9822. (2024) (Nov. 2018) (cit. on p. 29).

77. Strick, P. L., Dum, R. P. & Fiez, J. A. Cerebellum and nonmotor function. eng. *Annual Review of Neuroscience* **32,** 413–434. ISSN: 1545-4126 (2009) (cit. on p. 29).

78. Tee, J. & Taylor, D. P. A Quantized Representation of Probability in the Brain. *IEEE transactions on molecular, biological, and multi-scale communications* **5,** 19–29. ISSN: 2332-7804 (Oct. 2019) (cit. on p. 29).

79. Ullman, T. D., Goodman, N. D. & Tenenbaum, J. B. Theory Learning as Stochastic Search in the Language of Thought. *Cognitive Development. The Potential Contribution of Computational Modeling to the Study of Cognitive Development: When, and for What Topics?* **27,** 455–480. ISSN: 0885-2014. (2023) (Oct. 2012) (cit. on p. 30).

80. Ullman, T. D., Spelke, E., Battaglia, P. & Tenenbaum, J. B. Mind Games: Game Engines as an Architecture for Intuitive Physics. *Trends in Cognitive Sciences* **21,** 649–665. ISSN: 13646613. (2023) (Sept. 2017) (cit. on pp. 6, 30).

81. Van Rooij, I. The Tractable Cognition Thesis. *Cognitive Science* **32,** 939–984. ISSN: 1551-6709. (2023) (2008) (cit. on p. 25).

82. Vaswani, A. *et al. Attention Is All You Need* in *Advances in Neural Information Processing Systems* **30** (Curran Associates, Inc., 2017). (2023) (cit. on pp. 10, 15).

83. Veličković, P. *et al. The CLRS Algorithmic Reasoning Benchmark* June 2022. arXiv: 2205.15659 [cs, stat]. (2022) (cit. on p. 15).

84. Wang, K., Singh, R. & Su, Z. Dynamic Neural Program Embedding for Program Repair. *arXiv preprint arXiv:1711.07163.* arXiv: 1711.07163 (2017) (cit. on pp. 15, 26).

85. Windridge, D. & Thill, S. Representational Fluidity in Embodied (Artificial) Cognition. *Biosystems* **172,** 9–17. ISSN: 0303-2647. (2022) (Oct. 2018) (cit. on p. 30).

86. Wolf, T. *et al. Transformers: State-of-the-Art Natural Language Processing* in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (eds Liu, Q. & Schlangen, D.) (Association for Computational Linguistics, Online, Oct. 2020), 38–45. (2024) (cit. on p. 10).

87. Wolfram, S. *What Is ChatGPT Doing ... and Why Does It Work?* ISBN: 978-1-57955-081-3 (Wolfram Media, Incorporated, 2023) (cit. on p. 10).

88. Zečević, M., Willig, M., Dhami, D. S. & Kersting, K. *Causal Parrots: Large Language Models May Talk Causality But Are Not Causal* Aug. 2023. arXiv: 2308.13067 [cs]. (2024) (cit. on p. 31).

89. Zhang, H., Rich, P. D., Lee, A. K. & Sharpee, T. O. Hippocampal Spatial Representations Exhibit a Hyperbolic Geometry That Expands with Experience. *Nature Neuroscience* **26,** 131–139. ISSN: 1546-1726. (2023) (Jan. 2023) (cit. on p. 26).

90. Zhang, J. *et al. A Novel Neural Source Code Representation Based on Abstract Syntax Tree* in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (IEEE, Montreal, QC, Canada, May 2019), 783–794. ISBN: 978-1-72810-869-8. (2023) (cit. on pp. 26, 31).

Appendix A

Domain Specific Language (DSL)

The Domain Specific Language, including semantics and primitive types, as given by Fijalkow *et al.* [25].

1 Semantics

```
semantics = {
"empty": [],
"cons": _cons,
"car": _car,
"cdr": _cdr,
"empty?": _isEmpty,
"gt?": _gt,
"le?": lambda x: lambda y: x <= y,
"not": lambda x: not x,
"max": lambda x: lambda y: max(x, y),
"min": lambda x: lambda y: min(x, y),
"if": _if,
"eq?": _eq,
"*": _multiplication,
"+": _addition,
"-": _subtraction,
"length": len,
"0": 0,
"1": 1,
"2": 2,
"3": 3,
"4": 4,
"5": 5,
"range": _range,
"map": _map,
"iter": _miter,
"append": lambda x: lambda l: l + [x],
"unfold": _unfold,
```

```
    "index": _index,
    "fold": _fold,
    "is-mod": lambda x: lambda y: y % x == 0 if x != 0 else False,
    "mod": _mod,
    "is-prime": _isPrime,
    "is-square": _isSquare,
    "filter": lambda f: lambda l: [x for x in l if f(x)]
}
```

2 Primitive Types

```
    primitive_types = {
    "empty": List(t0),
    "cons": Arrow(t0, Arrow(List(t0), List(t0))),
    "car": Arrow(List(t0), t0),
    "cdr": Arrow(List(t0), List(t0)),
    "empty?": Arrow(List(t0), BOOL),
    "max": Arrow(INT, Arrow(INT, INT)),
    "min": Arrow(INT, Arrow(INT, INT)),
    "gt?": Arrow(INT, Arrow(INT, BOOL)),
    "le?": Arrow(INT, Arrow(INT, BOOL)),
    "not": Arrow(BOOL, BOOL),
    "if": Arrow(BOOL, Arrow(t0, Arrow(t0, t0))),
    "eq?": Arrow(INT, Arrow(INT, BOOL)),
    "*": Arrow(INT, Arrow(INT, INT)),
    "+": Arrow(INT, Arrow(INT, INT)),
    "-": Arrow(INT, Arrow(INT, INT)),
    "length": Arrow(List(t0), INT),
    "0": INT,
    "1": INT,
    "2": INT,
    "3": INT,
    "4": INT,
    "5": INT,
    "range": Arrow(INT, List(INT)),
    "map": Arrow(Arrow(t0, t1), Arrow(List(t0), List(t1))),
    "iter": Arrow(INT, Arrow(Arrow(t0, t0), Arrow(t0, t0))),
    "append": Arrow(t0, Arrow(List(t0), List(t0))),
    "unfold": Arrow(t0, Arrow(Arrow(t0, BOOL), Arrow(Arrow(t0, t1), Arrow(Arrow(t0, t0),
    ↪ List(t1))))),
    "index": Arrow(INT, Arrow(List(t0), t0)),
    "fold": Arrow(List(t0), Arrow(t1, Arrow(Arrow(t0, Arrow(t1, t1)), t1))),
    "is-mod": Arrow(INT, Arrow(INT, BOOL)),
    "mod": Arrow(INT, Arrow(INT, INT)),
    "is-prime": Arrow(INT, BOOL),
    "is-square": Arrow(INT, BOOL),
    "filter": Arrow(Arrow(t0, BOOL), Arrow(List(t0), List(t0))),
}
```

Appendix B

Experiment Hyperparameters

Hyperparameters	**Value**
α	0.3
β	0.7
γ	10
δ	150
ϵ	0.3
ξ	1 / 0.3
σ	1
`learning rate: generative model`	0.0001
`learning rate: forward policy`	0.0001
`e-steps`	2.000
`m-steps`	2.000
`epochs`	5
`batch size`	4
`inference steps`	100

Table B.1: Hyperparameters of both experiments.

Appendix C

Model Parameters

Class	**Parameter**	**Value**
IOEncoder		
	size_max	10
	d_model	512
RuleEncoder		
	d_model	512
Generative Model		
	d_model	512
	num_heads	8
	num_layers	2
	dropout	0.1
Forward Policy		
	d_model	512
	num_layers	2
	activation function	ReLU
Z		
	d_model	512
	num_layers	2
	activation function	ReLU

Table C.1: Model Parameters

Appendix D

Formal Grammars

Context-Free Grammars (CFGs) are essential in defining the syntactical structures of many formal languages. We can formalize the notion of CFGs as follows:

Let $G = (N, \Sigma, P, S)$ be a Context-Free Grammar, where:

- N is a finite set of non-terminal symbols.
- Σ is a finite set of terminal symbols with $N \cap \Sigma = \emptyset$
- P is a finite set of production rules, where each rule is of the form $N \to (N \cup \Sigma)^*$
- S is the start symbol, with $S \in N$

Given such a CFG, the derived sentence space $\Pi(G)$ is the set of all possible strings (or sequences of symbols) derivable from S.

Given a Context-Free Grammar G and a defined objective function f that maps any program $p \in \Pi(G)$ to a real value representing its desirability or fitness:

Find p^* such that:

$$p^* = \arg \max_{p \in \Pi(G)} f(p)$$

In other words, the problem is to locate a program p^* within the vast program space $\Pi(G)$ defined by G that maximizes (or, alternatively, minimizes) the objective function f.

A **Probabilistic Context-Free Grammar (PCFG)** is an extension of a CFG G, denoted as $G' = (N, \Sigma, P', S)$, where:

- N and Σ are as defined in the CFG.
- P' is a set of production rules, where each rule $A \to \alpha$ is associated with a probability $P(A \to \alpha)$, representing the likelihood of selecting that particular rule. These probabilities are subject to the condition that, for each non-terminal A, the sum of probabilities for all rules $A \to \alpha$ is equal to 1.

Appendix E

Levenshtein Distance

Given two strings s and t of lengths m and n respectively, the Levenshtein distance $d(s, t)$ is defined as the cost of the cheapest sequence of edits needed to transform s into t. The Levenshtein distance can be efficiently computed using dynamic programming. The idea is to construct a matrix where each cell (i, j) represents the cost of transforming the first i characters of s into the first j characters of t.

The formula for filling in the matrix is:

1. If $i = 0$, then $d(i, j) = j$ (cost of adding j characters).
2. If $j = 0$, then $d(i, j) = i$ (cost of deleting i characters).
3. Otherwise:

$$d(i,j) = \min \begin{cases} d(i-1,j) + 1 \\ d(i,j-1) + 1 \\ d(i-1,j-1) + \text{cost}(s_i, t_j) \end{cases}$$

   where $\text{cost}(s_i, t_j)$ is 0 if $s_i = t_j$ and 1 otherwise.

The value of $d(m, n)$ will then be the Levenshtein distance between s and t.

Appendix F

Trajectory Balance Loss

The following is a detailed derivation of the Trajectory Balance loss (for completeness sake), as described by Malkin et al. [**malkin˙trajectory˙2022**].

Combining Equation 1 and Equation 2 gives us:

$$\frac{R(s_T)}{Z_\theta} \prod_{t=0}^{T-1} \pi_\theta(s_{t+1}|s_t) = \frac{R(s_0)}{Z_\theta} \prod_{t=0}^{T-1} \beta_\theta(s_t|s_{t+1}) \tag{F.1}$$

Here β is the backward policy, predicting parent states. The initial state s_0 has the total flow and no reward, we can rewrite it and get:

$$Z_\theta \prod_{t=0}^{T-1} \pi_\theta(s_{t+1}|s_t) = R(s_T) \prod_{t=0}^{T-1} \beta_\theta(s_t|s_{t+1}) \tag{F.2}$$

We can now take the log on both sides:

$$\log\left(Z_\theta \prod_{t=0}^{T-1} \pi_\theta(s_{t+1}|s_t)\right) = \log\left(R(s_T) \prod_{t=0}^{T-1} \beta_\theta(s_t|s_{t+1})\right) \tag{F.3}$$

This can be rearranged to:

$$\log Z_\theta + \sum_{t=0}^{T-1} \log \pi_\theta(s_{t+1}|s_t) = \log R(s_T) + \sum_{t=0}^{T-1} \log \beta_\theta(s_t|s_{t+1}) \tag{F.4}$$

The trajectory balance loss is the squared difference:

$$\mathcal{L}_{TB} = \left(\log Z_\theta + \sum_{t=0}^{T-1} \log \pi_\theta(s_{t+1}|s_t) - \log R(s_T) - \sum_{t=0}^{T-1} \log \beta_\theta(s_t|s_{t+1})\right)^2 \tag{F.5}$$

In order to mitigate computational expense, I am embedding CFG rules rather than primitives,

such that each rule is unique. Thus, every predicted node has exactly one parent. Therefore, β will always be 1 and can be disregarded from the equation. Moreover, since we are solving tasks $x \in X$, we have a conditional reward distribution $R(s_T|x)$, as well as a conditional forward policy $\pi_\theta(s_T|x)$ and partition function $Z_\theta(x)$. This gives us the final loss:

$$\mathcal{L}_{TB} = \left(\log Z_\theta(x) + \sum_{t=0}^{T-1} \log \pi_\theta(s_{t+1}|s_t, x) - \log R(s_T|x) \right)^2 \tag{F.6}$$

Appendix G

Variational Inference

Variational Inference (VI) is a strategy that leverages a simpler distribution $q(z|x)$ to approximate a more complex target distribution. This is achieved by minimizing the Kullback-Leibler (KL) divergence, a similarity measure of probability distributions, between the true distribution and its approximation.

$$D_{KL}[q(z|x)||p(z|x)]$$

The approximation can be optimized by increasing the Evidence Lower Bound (ELBO) by:

$$\text{ELBO} = E_{q(z|x)}[\log p(x|z)] - D_{KL}(q(z|x)||p(z|x)) \tag{G.1}$$

Where $E_{q(z|x)}$ denotes the expectation under the recognition density q.

In the context of *amortized* VI, computation of the variational parameters ϕ is shared across multiple data points. Traditional VI might determine a unique set of variational parameters ϕ_i for each data point x_i, which is computationally intensive. Amortized VI, however, leverages a function (commonly a neural network) to compute the variational parameters ϕ for any data point x in a single pass, enhancing efficiency. In other words, we parameterize the recognition density.

this aligns with the free energy principle formulation [show that.] maximizing the evidence lower bound (ELBO), is equivalent to the negative free energy.

Amortized variational inference refers to the use of a parameterized function (e.g., a neural network) to approximate the posterior distribution in a variational Bayesian setting, where the parameters are learned once and can be used to infer the posterior for any given data point without retraining. It is "amortized" because the cost of learning the inference model is spread over all the data points it is used on.

GFlowNets relate to amortized variational inference in the sense that they learn a policy network that can generate samples for any given reward function without having to solve a new optimization problem for each sample. This is similar to how an amortized inference model can be applied to new data without additional optimization.

In both cases, the "amortization" allows for efficient inference or generation after the initial cost of learning the model. However, GFlowNets focus on learning to sample in proportion to a reward function, whereas amortized variational inference is concerned with approximating the posterior distribution of latent variables given observed data.