Hierarchical Bayesian Program Synthesis for Neural Algorithmic Reasoning

Botond Branyicskai-Nagy
Department of Computer Science
University College London

September 2024

Supervisors:
Prof Mirco Musolesi
Lorenz Wolf

Declaration of work undertaken

This report is submitted as part requirement for the *MSc Machine Learning* degree at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Botond Branyicskai-Nagy
September 2024

Abstract

Explaining a process algorithmically, given only input-output examples has been a long-standing core challenge of machine learning. Previous approaches introduce frameworks for inferring programs in a domain-specific language (DSL) that is usually a static set of manually designed primitives. This places a fundamental limitation on scalability and adaptability. This thesis introduces a novel framework for program synthesis that builds on the strengths of a transformer-guided search, a Bayesian inference scheme and neural library learning. We propose a method that integrates trainable neural primitives with a wake-sleep algorithm for improving generalisation in Programming-by-Example (PBE) tasks. By leveraging a transformer-based synthesis model, our approach addresses scalability and adaptability across domains. We show that this system allows the construction of DSLs dynamically, enabling more efficient and flexible program generation. Our primary contributions include the development of a system capable of finding general solutions to unseen tasks, facilitated by the use of a curriculum learning process. Initial experiments on a mixture of Boolean and arithmetic domains validate the feasibility of this approach, demonstrating successful discovery and reuse of compositional abstractions. Based on our findings, we further argue that there is significant potential for expanding the generalisation capabilities of neural models by inducing structured, compositional reasoning through a strong algorithmic prior.

Table of contents

Chapter 1

Introduction

Learning systems at a fundamental level work by manipulating some representation of their environment according to their inner model, optimising either or both of these based on some signal of their reliability. It is a guiding principle in designing artificial learners that they should share some characteristics with natural biological manifestations of a learning agent, the only examples being animals and ourselves, humans [1–4]. To understand them and try to replicate or simulate their behaviour in a way that is consistent across domains, is the subject of much of machine learning, either as a direct goal or implicit in the procedure or data used. As a way of modelling the brain's learning for its study, it is helpful for us to adopt the view of the brain as a biological computer [5] – or more specifically a computational theory of mind [6–8]. With this starting point, a potential path towards understanding it is immediately clear: the use of programs as inner representations of learnt behaviour.

Perhaps most importantly, programs allow the mind to simulate states, hypotheses and behaviour of just about any entity it has information about. Compressing the most salient patterns into a compact representation, programs can be accessed later to provide a model of the objects and processes around us. One of the first formulations of this hypothesis is found in Fodor's *The Language of Thought* [9] where the central premise of mental representations as structured entities – corresponding to objects, properties and states in the environment – is presented along with the idea of a formal grammar or set of syntactic rules for manipulating these. There is a specific advantage to programmatic representations that is the subject of this work: it equips the mind with the ability to reason about and explain its observations. To further illustrate this point, we now look at the role of structured, algorithmic reasoning in intelligence.

Reasoning with algorithms

Humans excel at applying step-by-step procedures or algorithms to explain phenomena around them. Take, for instance, a doctor diagnosing a patient. When a patient presents symptoms, the doctor likely cannot immediately guess the cause as they are unable to consider all possible diseases at once. Instead, they follow a structured reasoning process. Given data about symptoms, medical history and lifestyle factors they form hypotheses about potential conditions. As the doctor mentally runs through these options, they compare the symptoms against known patterns, evaluating the

likelihood of each condition and refining the diagnosis further until they arrive at the most likely explanation.

It poses a central challenge in machine learning to enable neural models to similarly learn these algorithms from data. Even the largest and most ambitious efforts in building artificial intelligence systems (such as Large Language Models) fail to perform this type of multi-step reasoning in a reliable and trustworthy manner [10–14]. But we need not look into such complex domains to see this gap – consider how young children effortlessly learn intuitive physics to catch a ball, and solve puzzles with ease before they know what combinatorics is. This capability emerges naturally in humans, yet, machine learning models often struggle with these tasks, as seen in challenges like ARC (Abstraction and Reasoning Corpus) [15, 16], where simple visual and algorithmic abstract reasoning performance of current systems remains about half that of the average human attempt [16]. Additionally, while neural networks can easily approximate functions over continuous spaces, tasks such as sorting, logic inference, or recursive problems require more structured representations. This is where the integration of symbolic components, formal languages and modular computational concepts becomes crucial.

Neural Program Synthesis as an Approach to Reasoning

Symbolic approaches have historically been outperformed by deep learning models in general, though it is also clear that some domains inherently require a constrained formal language that discretises and restricts the search space to only valid terms in a grammar (adhering to some syntactic and semantic rules). A straightforward example of this type of setting is mathematical reasoning, where most successful models employ a hybrid of neural networks and symbolic representations [17, 18].

Reasoning in a formal language has eluded deep learning models until recently, primarily because these models are designed to excel at tasks like perception and usually finding a local approximation of complex functions, where continuous representations are ideal. However, solving tasks that require structured, rule-based reasoning – such as mathematical problem-solving or algorithmic reasoning – demands discrete, modular representations that are not easily captured by traditional neural networks [19, 20]. While they are known for learning hierarchical representation (e.g. in computer vision [21]), deep learning architectures typically lack the strict compositionality and systematicity needed to represent formal logic, mathematical expressions, or procedural rules efficiently.

This gap has sparked renewed interest in approaches that combine the strengths of both neural networks and symbolic systems [18, 22]. These methods aim to leverage the learning capabilities of neural networks while incorporating a discrete, formal grammar. Given the vast search space, naive exhaustive search is infeasible (the number of combinations scaling exponentially with symbol vocabulary size) calling for more sophisticated exploration. For example, Neural Program Synthesis (NPS) has emerged as a promising approach, where deep learning is used to guide the search through a space of formal programs or logical expressions. Offering significant improvement on traditional search methods [23–25], NPS offers the potential to bring reasoning tasks within the reach of neural models.

In Bayesian Program Synthesis [26], which forms the foundation of our method, models search for programs that best explain the data while balancing complexity and likelihood. The Bayesian approach provides a principled treatment of approximate inference of latent variables conditioned on observed ones, while offering a natural framework for integrating neural networks to guide the search process to amortise costs. Models like DreamCoder [27] use this framework to infer programs from examples, with a recognition model guiding the search through a library of concepts – the set of objects and functions it builds algorithms from.

Concept Libraries

By treating the search for programs as a probabilistic inference problem, DreamCoder can generate concise programs using a set of learned concepts or building blocks which we will refer to as *primitives*. There is an initial set of these, which – crucially for our discussion going forward – is not learned but provided to the system by the engineer or user. These represent basic domain-specific operations (e.g. arithmetic or logic gates) and can be composed to form more complex programs. In the wake-sleep algorithm (originally from [28]) used by DreamCoder, the system alternates between synthesising programs (the wake phase) and refining its library of concepts based on observed programs (the sleep phase), allowing it to continually improve its program synthesis capabilities by effectively developing its own programming language.

An essential part of most program synthesis work is defining the Domain-Specific Language (DSL) in which the programs will be expressed (see Section 3.1.2 for a formal definition). The DSL is a constrained formal language that encodes the primitives (the basic building blocks from before), which the synthesis model can use to construct solutions. The design of a good DSL is crucial because it significantly influences the complexity of the search space. In fact, it is often the case that hand-designing a

DSL is the main contribution to a model's success [29, 30]. The algorithm relies on the engineer or user to identify basic operations relevant to the task at hand, such as addition or natural numbers in an arithmetic setting, recursive operators like `map` and `fold` for list-sorting or concatenation in a string manipulation domain.

1.1 The Generalisation Problem and a Hypothesis for Compositional Learning

Surprisingly, even large feed-forward and recurrent neural networks are known to be poor at generalising outside the training range in algorithmic tasks – even simple arithmetic such as counting or addition [19]. A number of approaches have emerged to tackle specifically this issue:

- Augmentation with logical and arithmetic primitives (akin to providing the model with access to a calculator) [31]. This may mean giving up differentiability of the proposal, although solutions exist where this is kept [32]. The advantage however is efficiency if the programs are algorithmically simple.
- Maintaining a purely neural architecture, but with a hand-designed unit that encourages algorithmic generalisation: see Neural Arithmetic Logic Units (NALU) [33] or Neural Arithmetic Units (NAU) [34] for addition/multiplication, or Neural Programmer-Interpreters [35] for a more general method that is able to learn the standard primary school algorithm for written addition (going digit by digit right to left, carrying the remainder to the next decimal) using a scratchpad for intermediate result storage .

In our approach, we take a third, seldom-explored path that goes against the traditional goals of efficiency and optimality. Instead, our focus is on developing trainable neural concepts or modules that can adapt to the data, irrespective of domain. This flexibility mirrors the way humans learn and generalise across tasks, where we identify the ability to store reusable, modular internal representations is key to achieving broad generalisation. Since human learning is the only example we have of a system that generalises effectively across a vast array of tasks, we strongly believe that creating neural systems that embody this structural prior offers the most promising route to achieving true, domain-agnostic reasoning.

While the two solutions above both guarantee better precision and scaling of the operators, they sacrifice the inherent plasticity and open-endedness that characterise

the brain's cognitive processes. Augmentations like these are not concerned with the fact that there is no evidence to suggest that human learners are equipped with these – or any other – algorithmic primitives at birth. Instead, we learn to construct (and refine) concepts through experience, which by extension serves as a guiding principle in our approach to designing an artificial reasoning system:

> *We hypothesise that by developing neural systems with inherent bias towards compositional, algorithmic internal representations and the ability to consolidate these modular concepts through experience, we can improve generalisation across different tasks and contexts. Specifically, we propose that training these concepts by following a curriculum, and constraining program search to valid compositions of primitives according to a set of formal syntactic rules, will help the system find solutions that generalise better to any reasonably expected input within the given context.*

To investigate this hypothesis, our model represents concepts in the form of neural programs (feed-forward networks) and is able to compose these controlled by a transformer-based synthesis architecture – a complete description of the method is presented in Chapter 4.

In our experiments, we observed that by progressing through a curriculum of input-output pairs generated by programs of increasing complexity, the model was able to learn and represent the fundamental concepts encoded, such as logic gates and basic arithmetic operations. When shown examples of data generated by simple composite functions of these primitives (e.g. NAND gate), the unobserved structure successfully discovered – in the form of source code – demonstrating the system's ability to integrate learned concepts and apply them in simple unseen configurations. These early outcomes are encouraging but require further validation and exploration to ensure broader applicability. While the results are preliminary, they provide early evidence that the system is capable of learning modular concepts and composing them in ways that reflect basic algorithmic reasoning, indicating a promising foundation for achieving more advanced, open-ended synthesis in future work.

1.1.1 Compositional Representations

To further motivate the choice of structural priors encoded in our approach, we argue below (after [21, 36, 37]) that the presence of higher-level compositional patterns in nature across a diverse set of domains serves as profound grounding for expecting efficient modelling of such patterns in the human brain.

Turing's living proof

Systems that exhibit successful learning in environments vastly more complex than themselves, by definition require some compression that captures a relevant set of features about their world. From the infinite pool of choices for this compression, it seems a daunting task – if not impossible – to settle on one algorithm for this purpose, yet it is apparent that our brains must be equipped with this capacity. The human brain is often taken as proof for the existence and feasibility of a computing mechanism capable of reducing its input signals and states to a internally manageable size while maintaining a remarkable amount of meaningful structure and detail. There is long-standing evidence that its capability extends beyond a fixed compression algorithm and is able to learn a new procedure based on the domain and context [38]. However, we have little more than hints of what exactly is done in a brain and classical algorithms developed for digital storage devices, however efficient, do not focus on universality and are by no means transferable to an arbitrary environment.

Computers equipped with modern programming languages are efficient and universal, in some ways more so than any biological machine. There are a multitude of software and specially designed algorithms available for a range of domains, yet nothing as general or adaptive as us humans, and *common sense* reasoning and acting remains elusive. As modern machine learning has shown, the question is more intricate than scaling up computational resources and dataset size, and rather a question of core mechanisms and architectures.

While trained deep learning systems may share some properties with how we think, at times making human-like mistakes, these observations are mostly superficial ones. As for its structure, a node in a neural network is a very crude and inaccurate model of a real neuron at best and it can be stated with relative certainty that the brain does not perform back-propagation to learn. One useful insight from deep learning is that hierarchical representations generalise well in a range of domains [39][21].

In the following we view latent neural structures through the more general lens of compositional representations, which involves constructing complex structures or concepts by combining simpler components or building blocks. In contrast with hierarchical representations, the focus will be on flexibility and reusability of smaller units to create more diverse and sophisticated representations. The human brain is adept at this form of processing, evident in how we understand language, create art, and solve problems – thus, it is the focus of many sub-fields in machine learning to try and replicate this aspect of intelligence.

1.2 Outline

Having stated our aims and motivation, introducing the context and the subject of our contributions, we move on to Chapter 2, where we review the key developments in the relevant fields, examining related work in neural program synthesis, hierarchical, Bayesian program synthesis, as well as neural algorithmic reasoning. Chapter 3 provides the necessary theoretical background, covering program representation through λ-calculus, domain-specific languages, and the STITCH algorithm for library compression, while also introducing HBPS more formally – this lays the groundwork for our approach. Building on this, Chapter 4 introduces our proposed method for Bayesian program synthesis of neural functions, detailing the integration of the newly introduced Primitive Inception phase into the the wake-sleep cycle, along with the attention mechanism used. Chapter 5 then presents an evaluation of the method, demonstrating its adaptation to domains such as Boolean logic, arithmetic functions, and mixed domains, with a focus on curriculum learning. Finally, Chapter 6 concludes the thesis with a summary of our contributions, a discussion of the limitations of our approach, and suggestions for further research directions. Our implementation used for experiments can be found in the Python repository[1].

[1] `https://anonymous.4open.science/r/bayesian_synthesis_neural_programs-0352`

Chapter 2

Related Work

As the method presented in this work combines recent research from a number of areas, it is important to clarify exactly how each of these contribute to, and differ from, our approach. Below is a survey of relevant topics and key results that have influenced the development of our framework. We begin by introducing the field of Neural Program Synthesis, followed by work on the Bayesian formulation of the problem with hierarchical priors, which together provide the main source of inspiration for our contribution. Subsequently, recent work in neural computation theory is reviewed with particular focus on functional programming and NTMs, and their connections to algorithmic reasoning. We argue that there is value in bringing these fields together to advance generalisation capabilities of synthesis models, as they share the common goals of algorithmic generalisation and scalability, and agree in the benefit of using programs as representations.

2.1 Neural Program Synthesis

Program Synthesis is an area of research that predates modern machine learning [40, 25] dealing with the problem of inferring an algorithm that explains a set of input-output examples. For example, given an unsorted list and its sorted version we aim to recover a generic sorting algorithm. At the heart of the classical framing of the task is combinatorial search: the space of possible programs constructed from the predetermined, fixed set of basic primitives has to be navigated such that a correct algorithm is found in reasonable time. These approaches, however, were often limited by the computational infeasibility of exploring the vast search space [25].

With renewed interest thanks to the success of deep learning guided search, a number of approaches have demonstrated promising performance [37, 27, 41]. Neural Program Synthesis [42]combines the principles of classical program synthesis with modern deep learning and Bayesian inference, to guide the search for correct programs more efficiently. Instead of relying solely on predefined rules and heuristics, neural models can learn patterns from data with minimal human involvement, allowing them to predict promising regions of the program space to explore.

One significant approach in this domain is the use of sequence-to-sequence models such as LSTMs, originally developed for tasks like machine translation, but adapted

to generate programs from input-output pairs [43]. These models treat the program synthesis problem as a sequence prediction task, where the input is the set of examples, and the output is the corresponding program represented as a sequence of tokens. Our model fits into this category, as it employs a transformer encoder-decoder architecture *translating* from input-output pairs to a source code language we formally define in Section 3.1

Another notable avenue of research is utilising reinforcement learning (RL) for program synthesis, mostly in combination with the previously mentioned techniques. Here, the agent is trained to maximise a reward signal that reflects the correctness of the generated program. By interacting with an environment (e.g. a program executor), the model learns to choose or write programs that produce the desired outputs, resulting in successful program generation strategies [44]. Incorporating LLMs in this process for writing code as well providing the agent with a curriculum is also common, with Voyager [45] showcasing the benefit of skill libraries in the form of short programs with remarkable success. Additionally, hierarchical and compositional approaches in RL-based methods [46, 44] have been shown to generalise better due to the narrowing of the search space as more complex structures are discovered.

A somewhat different paradigm, concerned with search over the space of a set of deep learning models is Neural Architecture Search (NAS) [47]. This approach is concerned with optimising the architecture itself of the neural network to better fit the training data. NAS has been shown to improve performance compared to fixed models by tailoring the network architecture to the specific characteristics of the problem on-the-fly [48]. However, this perspective is concerned mostly with technical performance gains by automating novel architecture discovery, while we will be concerned with a wider framework that encapsulates ideas from, but is not limited to NAS.

Despite recent advancements, challenges remain in ensuring that neural program synthesis models generalise well to unseen tasks, handle more complex program structures, and produce human-interpretable outputs. The following sections will explore how other methodologies, including Bayesian approaches and neural networks designed for algorithmic reasoning, address some of these challenges.

2.2 Hierarchical Bayesian Program Synthesis

Introduced in [26] and subsequently refined by Ellis et al. in [27] and [49], Hierarchical Bayesian Program Synthesis (HBPS) presents a scheme that integrates Bayesian inference with hierarchical structures to learn reusable subroutines and efficiently solve

complex tasks. The key idea behind HBPS is to model the synthesis process as a probabilistic inference problem, where the goal is to find the most probable program that explains the observed data.

As the name suggests, the space of possible programs is organised hierarchically here, with the simplest and most fundamental programs (initial primitives) at the lowest level and more complex compositions of these building blocks at higher levels. Bayesian inference is used to guide the search through this space, allowing the model to incorporate prior knowledge about e.g. program syntax and update its beliefs based on the likelihood of observed data.

DreamCoder is a seminal work in this area, and closest to our method in its procedure. It combined the principles of HBPS with deep learning to create a system that learns to solve new programming tasks by building on a library of previously learned subroutines [27]. The system alternates between *waking* phases, where it solves tasks using its current library, and *sleeping* phases, where it refines its library by identifying useful subroutines that can be reused in future tasks. The usefulness of a subroutine is determined by a score calculated as its *length* $\times$ *number of occurrences* – this step is called *Abstraction*. An efficiency improvement of this step was presented in [49], with a freely available Python library[1], which our method makes use of when finding abstractions (see Section 4.1.3).

The hierarchical nature of this approach allows for efficient exploration of the program space, as the model can focus on composing existing subroutines rather than generating programs from scratch. Additionally, the probabilistic framework provides a natural way to handle uncertainty and incorporate prior knowledge, making HBPS particularly well-suited for tasks where the program structure is complex or where data is scarce.

HBPS also offers significant advantages in terms of generalisation and interpretability. By building programs from a library of learned subroutines, the model can more easily transfer knowledge to new tasks, and the resulting programs are often more human-understandable than those generated by purely neural methods. This interpretability is crucial in applications where understanding the generated program is as important as its correctness.

[1] `https://github.com/mlb2251/stitch`

2.3 Differentiable and Functional Programming

Differentiable, as well as neural programming [32, 35, 31] is built on the principles of functional programming: a paradigm that uses functions as its main building block, composing them to write programs. Memory management is usually done with heaps, data is treated as immutable and type systems are also used. A core difference between imperative and functional programming is that common control flow patterns (e.g. recursive iteration) are abstracted over with higher-order functions (these take other functions as arguments) like map and fold. Functional programming also provides a strong theoretical foundation for composing programs in a modular and reusable manner, which aligns well with the goals of program synthesis.

Relevant to our context, differentiable and neural programming languages seek to leverage this compositional nature to design differentiable programs that can learn and execute complex programs. One approach is to design neural architectures that mimic the behaviour of functional programs, such as by using neural networks to represent algorithmic constructs [31]. This will be a crucial element in our method described below, as it allows almost arbitrary primitives to be trained, even on-the-fly if the domain is substantially changed.

By learning to compose these functions, the model can generate complex programs that solve a wide range of tasks, from data processing to symbolic reasoning. One of the key advantages of this scheme is its ability to produce programs that are both modular and interpretable. On one hand, the generated programs are easier to understand and debug, making them more suitable for applications where interpretability is important. As a consequence, the job of the synthesis model itself is simplified, as the search space becomes easier to traverse with the compositions (assuming the data itself is compositional, which is often the case in real world problems). This is hypothesised to result in better alignment of the embedding space of functions in the synthesis model and their meaning to a human observer.

For these reasons we observe that neural functional programming aligns well with the principles of hierarchical Bayesian program synthesis, where the goal is to learn reusable subroutines that can be composed to solve new tasks. Combining the strengths of HBPS for functional programming with neural networks, this framework promises the development of systems that are powerful and flexible, capable of solving complex tasks with a high degree of generalisation.

2.4 Neural Turing Machines

Neural Turing Machines (NTMs) are a class of neural networks designed to emulate the capabilities of a Turing machine by augmenting a neural network with an external memory component. Introduced by Graves et al. [50], NTMs extend the traditional neural network architecture by allowing the model to read from and write to a memory matrix, effectively enabling it to learn and execute algorithms that require the manipulation of data over multiple steps. The key innovation of NTMs is their ability to learn to control the read and write operations to the external memory, making them capable of solving tasks that require iterative processing, such as copying sequences, sorting lists, or performing simple arithmetic operations. This capability makes NTMs particularly relevant for program synthesis, with generated programs involving loops, conditional statements, and recursive function calls.

Differentiable Neural Computers (DNCs) are an extension of NTMs that further improve the model's ability to learn complex algorithms by introducing more sophisticated memory management mechanisms [23]. DNCs can dynamically allocate memory, manage memory usage more efficiently, and link different memory locations, enabling them to solve even more complex tasks.

In the context of program synthesis, NTMs and DNCs offer a way to learn programs that go beyond simple mappings from inputs to outputs. They can learn to execute sequences of operations that involve intermediate steps, making them suitable for tasks that require algorithmic reasoning. However, training these models is challenging due to issues such as gradient instability, difficulty in learning long-term dependencies, and the high computational cost of managing the external memory. As of yet, DNCs have only been demonstrated to handle simple tasks that conventional programming also solves easily.

Despite these challenges, NTMs and their variants represent a significant step toward integrating neural networks with traditional computational models. In cases where the programs involve intricate data manipulation and control flow, they may provide a better framework over functional languages, albeit at the cost of complicating the search procedure.

2.5 Neural Algorithmic Reasoning

Our final point of reference is the field of Neural Algorithmic Reasoning (NAR) [51–53], an emerging area of research that seeks to enable neural models to reason algorithmically,

allowing them to perform tasks that traditionally require explicit algorithmic steps, such as sorting, searching, or dynamic programming. A common approach is to use Graph Neural Networks (GNNs) to represent the underlying structure of the problem and to learn the algorithmic steps required to solve it. GNNs are particularly well-suited for this task because they can naturally model the relationships between different elements of the problem, such as the nodes and edges in a graph [54].

For example, Velickovic et al. [55] demonstrated how GNNs could be trained to approximate the behaviour of classical algorithms like breadth-first search (BFS) and shortest-path algorithms. By training the GNNs on a variety of graph-based problems, the model learns to generalise the algorithmic steps across different instances of the problem, effectively embedding the algorithm within the neural network.

NAR has significant implications for program synthesis. By embedding algorithmic reasoning within neural networks, these models can learn to perform tasks that involve complex reasoning over structured data, such as mathematical problem solving, symbolic manipulation, and even code generation [51]. This ability to learn and apply algorithms within a neural framework opens up new possibilities for combining the strengths of classical algorithms with the flexibility of neural networks. However, NAR also presents its own challenges, particularly in terms of learning algorithms that are interpretable, generalisable, and efficient. Ensuring that the neural network can capture the essential properties of the algorithm without overfitting to specific instances is a key area of ongoing research.

Summary of Related Work

These advancements in neural program synthesis, differentiable programming, and neural algorithmic reasoning have created a rich landscape of approaches for integrating neural networks with formal methods. The areas above have shown some of the most promising results in fields such as mathematical reasoning, code generation, and structured problem-solving. However, the integration of these methods into open-ended, adaptable and generalisable systems remains an open challenge. This work builds on these previous efforts by proposing a hybrid approach that combines Bayesian inference and neural primitives, to help bridge the gap toward human reasoning and alleviate some of the limitations identified in related work by offering more robust applicability across domains.

Chapter 3

Background

Having established the context of program synthesis and its intersection with neural learning in related work, this chapter delves into the theoretical foundations that underpin our approach. We first provide a detailed description program representation, beginning with the λ-calculus and DSLs, providing us with a formal framework for understanding how computations can be expressed and manipulated in our method. Building on these fundamental concepts we then discuss the technical details of Hierarchical Bayesian Program Synthesis for to aid our discussion of how our contribution extends it in Chapter 4.

3.1 Program Representation

3.1.1 λ-calculus

The λ-calculus [56] is a formal logic system for expressing computation based on function abstraction and application. It forms the foundation for much of functional programming and is widely used in program synthesis [24, 57, 27, 49] and representation for its clarity and universality – being Turing-complete. The core idea behind λ-calculus is that functions can be treated as first-class entities – meaning they can be passed as arguments, returned as values, or constructed dynamically. The three primary components or *terms* in λ-calculus:

1. Variables: represent input values to functions.
2. λ-abstractions: represent function definitions, in the form of $\lambda x.M$, where x is a variable and M is an expression.
3. Function applications: written as (MN), where M and N are both expressions.

Programs in λ-calculus are composed by applying functions to arguments, reducing expressions to their simplest form:

This process, known as β-reduction, eliminates function applications by substituting arguments into the function body, until no further reductions can be made, resulting in what is known as a normal form. If an expression has a normal form, β-reduction will eventually reach it, where no more substitutions or simplifications are possible.

The power of λ-calculus lies in its minimalism and expressiveness: with just function abstraction and application, it is possible to represent any computation[1]. This makes it a versatile intermediate representation in program synthesis tasks, where we seek to compose and manipulate functions in a systematic way.

In the context of program synthesis, λ-calculus provides a flexible foundation for constructing and manipulating neural program representations. By representing neural modules as λ-expressions, we can construct more complex programs through function composition and recursion. We implicitly use simply-typed λ-calculus [58] when performing data type management and function validation. It is an important detail that abstractions create new functions that act on variables specific to their scope and the body of the function itself. This structure enables more efficient search strategies by defining a compact space of possible programs that can be sampled, evaluated, and optimised.

3.1.2 Domain-specific languages

Any program synthesis framework is inevitably structured around some form of a domain-specific language (DSL). This defines the set of building blocks from which synthesis takes form. In our case, as we are dealing with λ-calculus as an intermediate representation, the DSL consists of functional expressions, each representing a neural network unit. Following [59] we now define the concept of a DSL along with an associated weight vector.

Definition 3.1.1 (DSL and Weight Vector ($\mathcal{D}$, ϕ))**.** *A domain-specific language (DSL) $\mathcal{D}$ is defined as a set of typed λ-calculus expressions. A weight vector ϕ for a DSL $\mathcal{D}$ is a vector of $|\mathcal{D}| + 1$ real numbers: one value for each DSL element $e \in D$, denoted ϕ, which controls the probability of element e being selected in a program. Additionally, ϕ_{var} controls the probability of a variable being selected in a program.*

Together with its weight vector, a DSL defines a probability distribution over programs, denoted $p(\rho|\mathcal{D}, \phi)$. This distribution can be understood by a generative sampling procedure that draws programs from $p(\rho|\mathcal{D}, \phi)$.

The utility of a DSL is twofold: it constrains the space of possible programs, making the search for valid solutions possible, and it encapsulates domain knowledge in the

[1]This is true for the untyped λ-calculus, which is Turing complete – meaning it can simulate any Turing machine. However, the typed λ-calculus imposes restrictions that can prevent certain forms of recursion or undefined behaviour, making it less expressive in some respects but safer and more predictable in terms of program correctness.

form of reusable functions. By defining a structured, limited set of building blocks, the DSL ensures that the synthesised programs are both syntactically correct and semantically meaningful within the domain.

In the context of HBPS, the DSL is not static—it evolves over time as the system learns new abstractions. As the wake-sleep cycle progresses, useful sub-programs are identified and abstracted into new functions that are added to the DSL, expanding its expressive power. This dynamic nature of the DSL enables the system to continually refine its library of functions, leading to more efficient and generalisable program synthesis over time.

The weight vector ϕ plays a crucial role in guiding the search process during synthesis. Higher weights indicate more likely components of a program, effectively biasing the search towards common patterns or frequently used primitives. This probabilistic approach to program generation, controlled by ϕ, allows the model to balance exploration (trying out less likely programs) and exploitation (favouring highly probable, reusable components).

As described in [59], this weighted DSL approach can dramatically improve the efficiency of program search in synthesis tasks, especially in domains requiring compositionality and reuse of learned concepts. By evolving the DSL and adjusting ϕ based on task performance, the system can tailor its search process to the specific needs of the domain, reducing the complexity of program generation.

3.1.3 Library compression through abstraction (STITCH)

Operating on λ expressions in a given DSL, STITCH [49] is a recent algorithm introduced for program synthesis that focuses on the efficient learning of reusable program abstractions – that is, define new functions across programs such that the overall code length is reduced. Previous methods in the program synthesis community [60, 27] have tackled this challenge by finding common fragments, or tree structures (in the computation graph), and refactoring existing programs with the abstractions.

STITCH, however, follows an approach the authors call *corpus-guided top-down synthesis*, which avoids the complexity of refactoring by synthesising abstractions from scratch. Unlike methods such as those presented in DreamCoder, which use semantics-preserving rewrite rules to expose shared structure across programs, STITCH directly searches for abstractions by examining syntactic patterns in the program corpus. The central idea is to guide the synthesis process towards abstractions that maximise shared structure, thereby enabling the discovery of reusable sub-routines that simplify the programs.

Compression as a Utility Function

The core objective is to minimise the size of a given corpus of programs by identifying abstractions that compress the programs efficiently. Following the work of [60, 27, 59], STITCH uses *compression* as the primary utility function for evaluating the quality of learned abstractions. Specifically, an abstraction is considered valuable if it reduces the overall size of the corpus when the programs are rewritten using it.

Formally, the utility function $U(A)$ for an abstraction A is defined as the product of the size of the abstraction and the number of places in the corpus where it can be applied. This balances two key properties of a useful abstraction:

- The **generality** of the abstraction, ensuring it applies to multiple locations within the corpus.
- The **specificity** of the abstraction, ensuring that it captures a significant amount of structure at each location.

The goal is to find abstractions that are both general enough to be reused frequently and specific enough to capture meaningful structure.

Learning Functional Abstractions

To illustrate this algorithm in action, consider the following set of programs written in the λ-calculus [49]:

```
λx.  *  8  (+ (*  3  1)  2)
λxs.  map (λx.  *  8  (+  4  (*  7  x))  xs
λx.  (- 9 (*  8  (+  x  (-  1  0)))
```

The algorithm's task is to identify and synthesise a functional abstraction that can compress these programs by capturing shared structure. The optimal abstraction found in this case is:

```
f0 = λα.  λβ.  (*  8  (+ α β))
```

This abstraction encapsulates the shared structure of the programs. When the programs are rewritten using `f0`, the resulting corpus is reduced:

```
λx.  f0  (*  3  1)  2
λxs.  map (λx.  f0  4  (*  7  x))  xs
λx.  (-  9  (f0  x  (-  1  0)))
```

By finding and applying this abstraction, STITCH can minimise the overall complexity of the programs while preserving their functionality. While this example only searches among three simple programs, the same algorithm applied to a larger corpus can identify relatively long abstractions, thus compressing their length, making synthesis down the line significantly easier.

3.2 Hierarchical Bayesian Program Synthesis

In the following we introduce the framework used in the Bayesian formulation of the program synthesis problem.

3.2.1 Probabilistic Model

To perform inference in the space of programs, we first need to develop our probabilistic model of the generative process that produced our observations. Presented in Fig. 3.1, our graphical model consists of the following elements: a shared library $\mathcal{L}$ containing functions $f_0, \ldots, f_{L-1}$, a set of programs $\mathcal{P} = \{\rho_1, \ldots, \rho_n\}$ directly constructed from (and thus dependent on) $\mathcal{L}$ and a set of independent tasks $\mathcal{T} = \{t_1, \ldots, t_n\}$ given by sampling inputs and passing them through the corresponding program to get outputs. We assume independence of tasks as they are, by definition, each generated by a corresponding program sampled from the library. The manner in which the programs themselves are sampled may depend on the dataset, but in every case they only depend on the true library of primitives and there is no mechanism by which they would interact with the other programs. These observations justify the choice of graphical model we will use for inference going forward.

The objective will be to maximise the posterior distribution $p(\mathcal{P}, \mathcal{L} \mid \mathcal{T})$ and so we first write down the joint, factorised over tasks $i \in \{1, \ldots, n\}$ according to our model as

$$p(\mathcal{L}, \mathcal{T}, \mathcal{P}) = p(\mathcal{L}) \prod_{i=1}^{n} p(\rho_i \mid \mathcal{L})\ p(t_i \mid \rho_i)$$

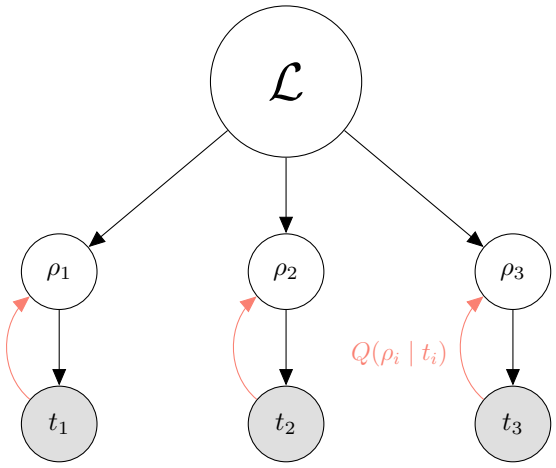

Fig. 3.1 Graphical model of the hierarchical Bayesian program learning setup. Nodes t_i represent the input-output examples, or tasks, while ρ_i are the programs that generate these tasks. The central node $\mathcal{L}$ is the shared library of reusable functions or subroutines. All t_i are treated as observed variables throughout training. The model is trained using an iterative wake-sleep procedure: **Wake phase:** Given the library $\mathcal{L}$ and the observed tasks t_i, the program synthesis network Q generates candidate programs ρ_i. These programs are sampled from the posterior distribution $p(\rho \mid t, \mathcal{L})$ and are optimised to maximise the likelihood of generating correct solutions for the observed tasks. **Sleep phase:** Given the proposed programs generated during waking, we update the library $\mathcal{L}$ through a process of abstraction and compression, where frequently occurring program structures are incorporated as new reusable functions. The model then simulates new programs by sampling from $p(\rho \mid \mathcal{L})$ as well as taking inputs from t. These samples are used to re-train the synthesis network $Q(\rho \mid t)$ to approximate the posterior by learning from both the true programs (from waking) and the fantasised programs (from sleeping).

or for ease of numerical handling the log-joint as

$$\log p(\mathcal{L}, \mathcal{T}, \mathcal{P}) = \log p(\mathcal{L}) + \sum_{i=1}^{n} (\log p(\rho_i \mid \mathcal{L}) + \log p(t_i \mid \rho_i))$$

which follows from the conditional and marginal independencies encoded in the graphical model in Fig. 3.1. We thus identify three components:

- The library prior $p(\mathcal{L})$: our belief about which subroutines or functions are more likely to be useful for explaining the data, before any observations are taken into account. By Occam's razor we will typically favour simplicity here, and in general diversity has also been shown to be beneficial in identifying good primitive features [61]. This is implicitly encoded in our approach, by enforcing the syntax as well as in the way the curriculum is structured (see Section 5.3).

- The program likelihood $p(\rho_i \mid \mathcal{L})$ reflects the probability of generating a given program using the functions contained in $\mathcal{L}$. Shorter compositional programs are more likely under this distribution as we will see.
- The task likelihood $p(t_i \mid \rho_i)$ is the probability of observing the given input-output examples t_i had program ρ_i been used to generate them. This usually involves executing the proposed ρ_i on the inputs and comparing its results with the true outputs.

3.2.2 Wake-Sleep Cycle

DreamCoder [27] structures the learning process in a wake-sleep cycle, originally introduced as expectation-maximisation in the Helmholtz machine [28]. At a high level, the synthesis step itself is performed in the wake phase, where proposals $\hat{\mathcal{P}}$ are generated conditioned on the input-output pairs $\mathcal{T}$. These are ranked according to their approximate likelihoods under the recognition model $Q(\rho_i \mid t_i)$. During the sleep phase, a probabilistic generative model is trained which defines the prior as well as the network $Q(\rho \mid t)$ outputting the approximate posterior of a program. Next, to expand on these stages further, we reiterate the key elements while referring the reader to Fig. 3.1 providing intuition before we proceed to write down the corresponding iterative updates for each step, working to maximise a lower bound on the posterior over $\mathcal{L}$ given tasks $\mathcal{T}$.

Wake phase. Given the library $\mathcal{L}$ and the observed tasks t_i, the synthesis network Q generates candidate programs ρ_i for each task, ranked by their likelihood. $Q(\rho \mid t)$ is an approximation for the posterior distribution $P(\rho \mid t, \mathcal{L})$ optimised (during *sleeping*) to maximise the likelihood of generating correct solutions for the observed tasks. In our method, this amortised sampling step is performed by a transformer-based synthesis model, defined in Section 4.1.2. The corresponding maximisation step is:

$$\rho_t = \arg\max_{\rho} p(\rho|t, \mathcal{L}) \propto p(t|\rho)p(\rho|\mathcal{L}), \quad \forall t \in \mathcal{T}$$

where the program proposal ρ_t for task t is chosen such that it maximises the likelihood.

Abstraction phase. Equipped with the generated proposals, we proceed by abstracting over common substructures in the code, adding them to $\mathcal{L}$. This step has been improved since DreamCoder's publication, making STITCH [49] the preferable

framework for compressing λ-calculus expressions efficiently. In this step we update the library such that is maximises the joint:

$$\mathcal{L} \leftarrow \arg\max_{\mathcal{L}} p(\mathcal{L}) \prod_{t \in \mathcal{T}} \max_{\rho \in \rho_t} p(t|\rho) p(\rho|\mathcal{L})$$

where $p(\mathcal{L})$ is our prior over libraries and $p(t|\rho)$ is the likelihood of a task $t \in \mathcal{T}$ given program ρ (see Section 3.2.1). The maximum task likelihood is taken over program proposals ρ_t, for each task t.

Dreaming phase. With $\mathcal{L}$ updated, the model needs to be re-trained to recognise potential applications of the added abstractions. This is done by drawing programs at random from the new $P(\rho \mid \mathcal{L})$ (without conditioning on observed tasks) and executing these to create synthetic data referred to as *fantasies*. These can then be used to re-train the synthesis network $Q(\rho \mid t)$ without the need for more real data about the abstractions from the previous step. Equivalently we write

$$\theta \leftarrow \arg\min_{\theta} J\left(Q_\theta(\rho|t) - p(\rho|t, \mathcal{L})\right),$$

where the synthesis network parameterised by θ is trained to minimise an objective J given data $t \sim \mathcal{T}$ (*replay*) or $t \sim \mathcal{L}$ (*fantasy*).

Having discussed the wake-sleep cycle, we introduce the other crucial component of our method, the transformer architecture responsible for synthesising program generations.

3.3 Encoder-Decoder Transformer

The *Encoder-Decoder Transformer* architecture, introduced by Vaswani et al. in [62], has revolutionised sequence-to-sequence tasks by replacing recurrent models with an attention-based architecture. It consists of two main components: an *encoder*, which processes the input sequence, and a *decoder*, which generates the output sequence. Each of these is built using self-attention layers (Section 3.3.2) and feed-forward networks, enabling efficient processing of long-range dependencies in parallel.

3.3.1 Tokenisation and Embeddings

Before the input is passed into the encoder, it must first be tokenised. This involves converting the input sequence into discrete tokens (integers), which are then mapped

into *embeddings*, i.e., dense vectors that represent the tokens in a continuous space. In essence, embeddings allow the model to capture semantic information about the tokens.

Key aspects of this process include:

- **Pad Token**: Sequences often need to be padded to a uniform length to fit into batches. The pad token is used to fill these extra positions, which are masked during attention to prevent the model from processing them.
- **End Token**: To signal the end of a sequence, an end-of-sequence token (usually written `<eos>`) is appended. This allows the decoder to know when to stop generating further tokens.
- **Positional Encoding**: Since transformers lack the sequential inductive bias of RNNs, they need a way to encode the order of the tokens. This is achieved using *positional encoding*, which adds information about the position of each token in the sequence to its embedding. These are usually sinusoidal functions that vary based on the token's position, but can also be a raw index, allowing the model to distinguish between tokens at different locations in the sequence.
- **Masking**: Masking is critical for ensuring that the model does not attend to certain tokens during training. For example, the pad tokens are masked to prevent their influence on learning, and during decoding, a causal mask is used to ensure that the decoder only attends to tokens that have been generated so far, preventing the model from "cheating" by looking at future tokens.

3.3.2 Attention Mechanism

At the core of the transformer architecture is the *attention mechanism*, which allows the model to focus on different parts of the input sequence while processing each token. The attention mechanism is divided into two main parts: *self-attention* and *cross-attention*.

Self-Attention

Self-attention is used within both the encoder and the decoder. It allows each token in a sequence to attend to every other token, enabling the model to capture long-range dependencies. The self-attention mechanism works by computing three vectors for each token: a *query* vector, a *key* vector, and a *value* vector.

The attention score for each token is computed as the dot product of the query vector of one token with the key vectors of all other tokens, followed by a softmax operation to normalize the scores. The final output is a weighted sum of the value vectors, with the weights determined by the attention scores.

The formula for self-attention can be expressed as:

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

Where: Q is the query matrix, K is the key matrix, V is the value matrix, d_k is the dimensionality of the key vectors, and the softmax ensures that the attention scores sum to 1.

Cross-Attention

In the decoder, *cross-attention* is used to allow the decoder to focus on the relevant parts of the encoder's output when generating the next token in the sequence. Cross-attention works similarly to self-attention, with the same formula as self-attention, but now the query Q comes from the decoder's previous output, and the key K and value V matrices come from the encoder. This allows the model to align the decoder's tokens with the relevant parts of the encoder's context, thus improving the accuracy of the output sequence.

This architecture and attention mechanism make the transformer highly efficient at learning and generalising from sequence data. It allows the model to handle long sequences with ease, making it suitable for a variety of tasks, such as machine translation, text generation, and – most importantly for our purposes – can be adapted for program synthesis, as presented in Algorithm 1.

3.3.3 Multi-Head Attention

The *Multi-Head Attention (MHA)* mechanism is a core component of the transformer architecture. Instead of computing a single attention score for each token in a sequence, multi-head attention splits the input into multiple "heads." Each head performs attention independently, focusing on different parts of the sequence. This allows the model to capture a wider range of dependencies and relationships between tokens. After each head computes its attention score, the results are concatenated and projected back to the original size. Formally, we write:

$$\text{MHA}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W_O$$

where each head_i is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Here, W_i^Q, W_i^K, and W_i^V are projection matrices for the query, key, and value of each head, and W_O is the output projection matrix. This mechanism allows the transformer to learn more contextual representations by simultaneously attending to different positions within a sequence, capturing more intricate patterns.

Algorithm 1 Encoder-Decoder Transformer with Multi-Head Attention

Require: $\mathbf{x} \in \mathbb{R}^{B \times T_x}$: Tokenised input sequence (e.g., input-output task)
Require: $\mathbf{y} \in \mathbb{R}^{B \times T_y}$: Tokenised target sequence (e.g., target source code)

function SynthesisModel($\mathbf{x}, \mathbf{y}$)
  $\mathbf{X}_e \leftarrow \text{Embedding}(\mathbf{x}) + \text{PositionalEncoding}(\mathbf{x})$
  $\mathbf{Y}_e \leftarrow \text{Embedding}(\mathbf{y}) + \text{PositionalEncoding}(\mathbf{y})$
  $\mathbf{M}_x \leftarrow \text{SourceMask}(\mathbf{x})$ ▷ Attention mask for $\mathbf{x}$
  $\mathbf{M}_y \leftarrow \text{TargetMask}(\mathbf{y})$ ▷ Attention mask for $\mathbf{y}$ (including future masking)

  ▷ Encoder: Apply self-attention to $\mathbf{X}_e$
  for $l = 1$ to L_{enc} **do**
    $\mathbf{Z} \leftarrow \text{MHA}(\mathbf{X}_e, \mathbf{M}_x)$ ▷ Multi-Head Attention
    $\mathbf{Z} \leftarrow \text{LayerNorm}(\mathbf{Z} + \mathbf{X}_e)$
    $\mathbf{Z} \leftarrow \text{FFN}(\mathbf{Z})$ ▷ Feed-Forward
    $\mathbf{X}_e \leftarrow \text{LayerNorm}(\mathbf{Z} + \mathbf{Z})$
  end for

  ▷ Decoder: Apply self-attention and cross-attention
  for $l = 1$ to L_{dec} **do**
    $\mathbf{A} \leftarrow \text{MHA}(\mathbf{Y}_e, \mathbf{M}_y)$ ▷ Masked self-attention within $\mathbf{y}$
    $\mathbf{A} \leftarrow \text{LayerNorm}(\mathbf{A} + \mathbf{Y}_e)$
    $\mathbf{C} \leftarrow \text{MHA}(\mathbf{A}, \mathbf{X}_e, \mathbf{M}_x)$ ▷ Cross-attention with $\mathbf{x}$
    $\mathbf{C} \leftarrow \text{LayerNorm}(\mathbf{C} + \mathbf{A})$
    $\mathbf{Y}_e \leftarrow \text{FFN}(\mathbf{C})$ ▷ Feed-Forward Network
    $\mathbf{Y}_e \leftarrow \text{LayerNorm}(\mathbf{Y}_e + \mathbf{C})$
  end for

  $\mathbf{P} \leftarrow \text{Softmax}(\mathbf{W}_o \mathbf{Y}_e)$ ▷ Projection to output space: unembedding matrix $\mathbf{W}_o$

  return $\mathbf{P}$
end function

Chapter 4

Bayesian Program Synthesis of Neural Functions

The method developed here is built on a mixture of previous approaches to the Programming-by-Example (PBE) problem with the specific aims of scalability, adaptability and compatibility with a wide range of domains. Its novelty lies in connecting a number of promising areas of research, and can be summarised as curriculum-based [63] wake-sleep Bayesian algorithm learning [27] with neural primitives [32], with a transformer-based generative model for synthesis. Our primary contributions are the combination of Bayesian library learning and purely neural primitives, and the application of an encoder-decoder transformer architecture [62] for inference of programs in a wake-sleep scheme.

The main strengths of this approach as compared to other work on program synthesis are twofold: in comparison to DreamCoder [27] the neural primitives in principle allow for domain-agnostic generalisation, and the transformer-based synthesis model is more efficient than traditional search-based approaches [57]. The ability to learn arbitrary neural mappings is introduced for unseen functions where synthesis has failed. This framework allows the model to develop its DSL given new tasks as well as combine different domains when performing abstraction.

In the following each model choice will be discussed in detail, beginning with the extended wake-sleep Bayesian algorithm learning framework, which forms the backbone of the synthesis process.

4.1 Bayesian Inference of Programs

To connect the regimes of the λ-calculus introduced in Section 3.1.1 and the neural building blocks of our approach, we first define the custom-built compiler, then describe the contributions to the wake and sleep phases respectively, in more detail.

4.1.1 Neural Function Compiler

Before we can apply the HBPS framework to our neural concept library, we need a principled way of handling the modular primitives at execution time. For this

purpose, we implement a procedure that constructs a neural network from the library of primitives (themselves neural units). Note that the weights of this network will have already been optimised when the individual modules were trained (see Section 4.1.3) and thus only needs to be assembled. To perform this task, a custom-built compiler is used, which as input takes an internal source code string to output a neural network. The source code language (see example in Fig. 4.1) is a set of instructions specifying which block (denoted $\{\texttt{f0}, \texttt{f1}, \dots\}$) to apply to which input vector (variables $\{\texttt{i0}, \texttt{i1}, \dots\}$) or intermediate result vector (variables $\{\texttt{v0}, \texttt{v1}, \dots\}$). We use parentheses to make function arguments visible, and the new line character to separate statements for clarity. The last line is always a return statement, with a semicolon indicating the end of the program.

The reader may note that the example in Fig. 4.1 includes variable assignments; at first glance this suggests we are no longer using a purely functional programming paradigm as statements like these are characteristic of imperative languages. However, these variables are only used for temporary memory allocation to allow easier routing and scope management, and are never allowed to be assigned if they are not either used by a subsequent function call or returned. General variable assignment and keeping track of global objects would lead to an altogether more complex search space of programs, and we would no longer be able to apply the abstraction search introduced in Section 3.1.3. For this reason, we constrain our source code language's syntax for variable assignments specifically, such that it remains purely functional in nature.

With this compiler defined, we are able to integrate neural primitives with HBPS, which will be the main subject of the coming sections. To begin with, we further detail the contributions introduced in Section 1.1, and explain how we extend the framework of DreamCoder developed in Section 3.2.

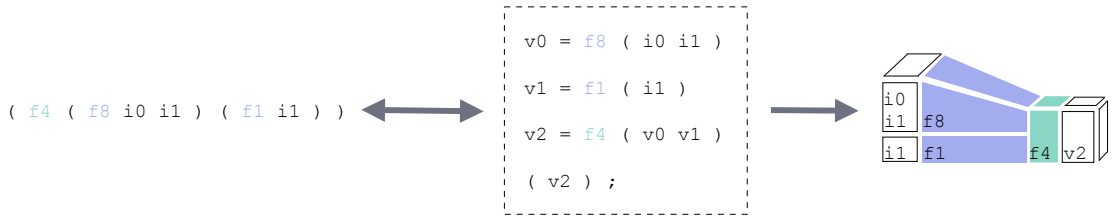

Fig. 4.1 Functional program representations: (left) λ-term: used for *Abstraction*; (middle) internal source code: allows variable management and construction of neural function; (right) neural blocks: program as a computational pipeline built from primitives `f1, f4, f8`, input vectors `i0` and `i1`, output vector `v2`. The intermediate values `v0, v1` are passed on immediately to the next block (`f4` in this case).

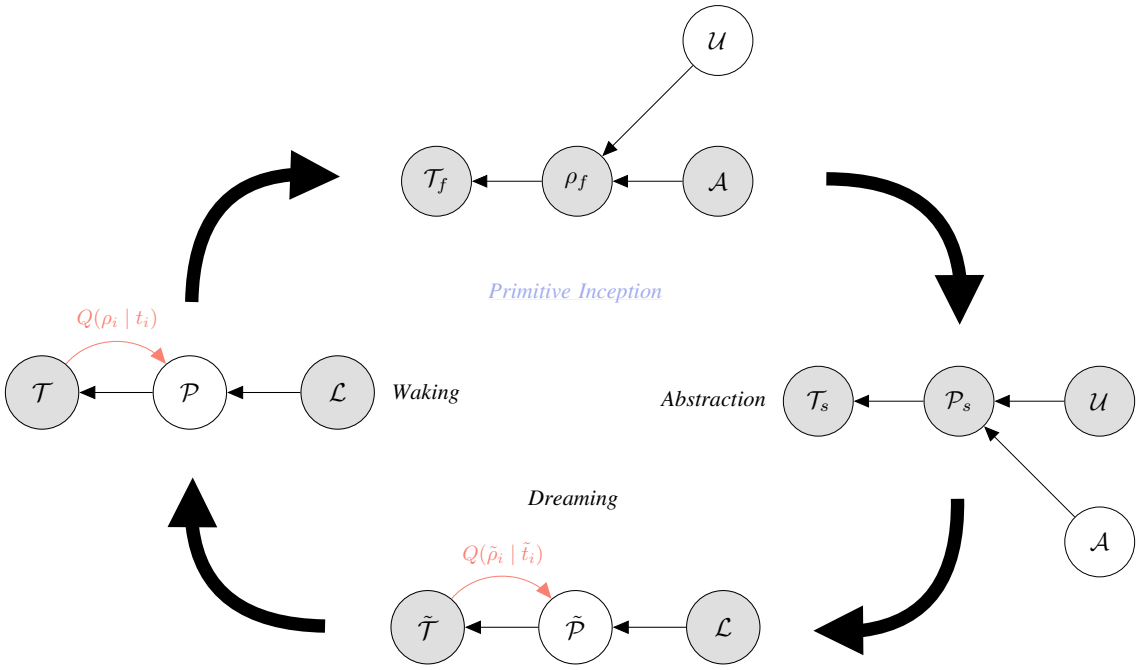

Fig. 4.2 The main wake-sleep cycle for synthesis of programs $\mathcal{P}$ and building the concept library $\mathcal{L}$. Starting with a *Waking* phase, given the observed (shaded) tasks $\mathcal{T}$ we infer the latent (not shaded) programs $\mathcal{P}$ with the synthesis model Q, building from $\mathcal{L}$ (also treated as an observed variable at this stage). Next, based on the outcome of the synthesis, we split the input-output pairs into failed ($\mathcal{T}_f$) and successful ($\mathcal{T}_s$) sets of tasks, as well as saving the successful programs $\mathcal{P}_s$. These move forward to the *Sleeping* phase, first with the *Primitive Inception* step, making use of $\mathcal{T}_f$, and training a new primitive on these tasks. This amounts to inference of the now unobserved set of neural primitives $\mathcal{U} \subseteq \mathcal{L}$, where ρ_f is the one line of source code calling the new primitive function. Independently from this process, we also perform *Abstraction*, where we compress $\mathcal{P}_s$ by finding useful sub-routines that repeat often and saving them to the set of abstractions $\mathcal{A} \subset \mathcal{L}$ in the form of λ-terms in the context of the DSL defined by $\mathcal{L}$. The synthesis model Q has to be familiarised with this newly updated library, which is the goal of the *Dreaming* stage: sampling from the $\mathcal{L}$ the model generates 'fantasies' $\tilde{\mathcal{P}}$, which are synthetic programs aimed to teach Q how to use the new library. These programs are then used to produce a set of fantasised tasks $\tilde{\mathcal{T}}$ by executing them on random inputs to obtain outputs. This allows supervised training of Q to learn a mapping from $\tilde{\mathcal{T}}$ to $\tilde{\mathcal{P}}$, or in other words, to approximate $p(\tilde{\mathcal{P}} \mid \tilde{\mathcal{T}}, \mathcal{L})$. This concludes the *Sleeping* phase, with $\mathcal{L}$ and Q passed on to the next wake-sleep cycle.

4.1.2 Wake Phase with Attention Mechanism

The goal of the wake phase is to generate new suitable programs for solving a given task. More specifically, given a library given a *Library* $\mathcal{L}$ of neural concepts (or functions)

and input-output examples (tasks) $\mathcal{T}$ the model synthesises a plausible set of programs $\mathcal{P}$. $\mathcal{L}$ contains reusable functions that can be composed to solve a given task. For instance, in the domain of simple arithmetic operations, it might include functions such as addition, subtraction, multiplication and division. Guiding the synthesis of programs is the training data in $\mathcal{T}$, consisting of sets $t = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of pairs of inputs x_i to a single program ρ and their corresponding outputs $y_i = \rho(t)$. For example, consider $t_+ = \{(2, 3) \to 5, (4, 5) \to 9, \ldots\}$ generated by $\rho_+(x) := x_0 + x_1$. While the model relies on input-output examples in $\mathcal{T}$, the supervision is not as direct as in classical supervised learning. This step can be understood as *weakly supervised*. The examples provide a general direction, but the actual program synthesis involves exploring various possible programs that are consistent (approximately) with the examples – defined as the following:

Definition 4.1.1 (Approximate Program Consistency)**.** *A program ρ is said to be approximately consistent with a set of examples in the task set $t \in \mathcal{T}$, where $t = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ if, for all $(x_i, y_i) \in t$, the absolute difference between the program's output and the desired output is within a specified tolerance $\epsilon(t) \in \mathcal{R}$. Formally, we say that ρ is approximately consistent with t if and only if:*

$$|\rho(x_i) - y_i| \leq \epsilon(t) \quad \forall\ (x_i, y_i) \in t,$$

where ϵ is, in general, a function of t and represents the allowable error margin for the example set.

In some settings – such as systems working with exact primitives – the threshold ϵ will be zero, with no tolerance for error: we can be sure that if there is any error, the program must be incorrect as the true or generating choice of subroutines guarantees full consistency. This is not the case for our neural primitives however, as they are always approximations of the true exact primitives that generated the data.

Transformers for example-to-code translation

Only having access to this signal of consistency and no true labels makes the supervision less explicit and more indirect. This softly supervised nature of the problem requires blending elements of supervised learning with exploratory or search-based methods.

In our approach we employ an attention mechanism that is most akin to an encoder-decoder transformer [64, 62] used in machine translation – this is shown in Algorithm 1. This choice is motivated by two main properties of transformers: their ability to handle

the varying length of task and source code sequences, and the attention mechanism [64] providing a fitting way of on-the-fly function retrieval based on the task and context. The former is an evident reason for the widespread applicability of these models, and so let us see how the latter fits in to our framework. The central observation is that the problem of program synthesis can be posed as one of translation: the goal is to obtain a sequence of tokens Y in the *language* representing source code, given the tokens of task t, encoded as X.

Note that the vocabularies (set of allowed tokens) of the tasks and code are not the same, as the former, written $\mathcal{V}_X$, may include any integer, separator (`_`, `-`, `;` etc.) or letter, while the latter, $\mathcal{V}_Y$, only holds functions `f*`, inputs `i*`, intermediate variables `v*` (where * is an integer) and the separators required by the syntax: `(`, `)`, `;`, `newline`, `=`. This however does not pose a challenge to the transformer seen in Algorithm 1, as we generate code tokens one by one, while attending to a fixed length of the input-output sequence (determined by the context window, see Table A.2 for our settings). We use a padding (`_`) and an end token (`;`) (refer to Section 3.3) as is standard to make training more stable and efficient.

4.1.3 Sleep Phase with Primitive Inception Step

Previous approaches [27, 57] consider a sleep phase with two sub-stages: the *Abstraction* process attempts to expand $\mathcal{L}$ with new composite functions constructed from the DSL with the aim of compression; finally, the model $Q(\rho|x)$ is trained to synthesise programs with the newly updated $\mathcal{L}$ – this is called the *Dreaming* step [28].

Our work extends this framework with the *Primitive Inception* phase where, before abstractions are drawn, the data from failed tasks during waking is used to train a new primitive and add it to $\mathcal{L}$. This is a crucial step for adapting to arbitrary domains, and is our key contribution to the HBPS scheme.

Primitive Inception. This stage of the sleep phase uses the inputs and outputs from failed tasks $(X_f, Y_f) \subseteq \mathcal{T}_f$ – that is, where synthesis did not produce an approximately consistent program – to train a new neural network, and add it to the set of primitives in $\mathcal{L}$. By training the synthesis model to use this new concept, in the Dreaming phase, the next time similar examples are seen this task is more likely to be solved.

There are a number of choices to make regarding the guardrails on this process. It is clear from the nature of gradient-based optimisation of feed-forward neural networks that they are not data-efficient and can easily overfit if only a small number of training

samples are available. To overcome or at least limit this behaviour we make a number of adjustments to the default training procedure.

First, we introduce a lower threshold on the loss which, if not reached after a predetermined number of epochs, the weights are re-initialised (using Xavier initialisation [65]) to mitigate getting stuck in local minima. Similarly, if the loss is stagnating too long we again reset the weights. Training is retried a specified number of times, or until the threshold is reached. This corresponds to the error margin tolerance ϵ introduced in Definition 4.1.2 of approximate consistency.

Second, the dimensions of the neural unit – parameterised by hidden layer depth and width – are incremented upwards by a factor of 2 every few steps (typically 2-10). By Occam's razor this aims to ensure that if a simpler architecture is sufficient, it will be preferred.

The complete wake-sleep cycle is illustrated in Fig. 4.2, with our contribution (Primitive Inception) highlighted in colour. Finally, Algorithm 2 presents the procedure step-by-step, with the four steps in total acting to update the synthesis model Q and refine the library $\mathcal{L}$ given input-output pairs X, Y. We refer the reader to Section 3.2.2 for definitions of the Abstraction and Dreaming phases.

Algorithm 2 Main Cycle of Wake-Sleep Algorithm

```
1: procedure WakeSleep(Q, X, Y, L, rounds)
2:     for i ← 1 to rounds do
3:         Wake Phase
4:         proposals, failed_tasks ← Wake(Q, X, Y, L)
5:
6:         Sleep Phase
7:         L ← Abstraction(proposals)
8:         L ← Primitive Inception(failed_tasks)
9:         Q ← Dreaming(X, Y, L)
10:    end for
11: end procedure
```

4.2 A Growing Library of Neural Functions

In the following we discuss the key implications of using a neural library to the overall method and experimental considerations. We consider in particular the role of a curriculum-based training procedure, where each wake-sleep cycle, the dataset of tasks provided gets progressively harder (see Section 5.3 for details).

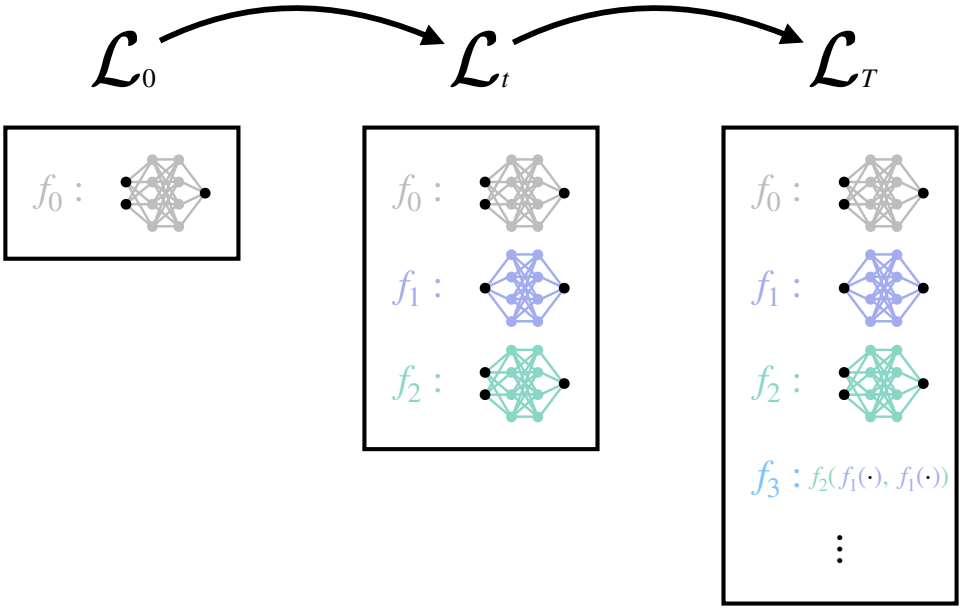

Fig. 4.3 Library $\mathcal{L}$ of neural concepts at three different time-steps: initialisation, during learning (t) and final state (T). Up to t, the library is expanded with new functions when the current set of functions (just f_0 at $\mathcal{L}_0$) is found to be insufficient to explain an example: in this case a new network (e.g. f_1) is instantiated and optimised for that example and added to $\mathcal{L}$. From t to T, equipped with more primitives, we observe another way to extend $\mathcal{L}$, by abstraction: the set of generated programs in the wake phase are compressed, by finding subroutines (like f_3 above) that appear often.

Note on interpretability. It could be argued that the main product of the complete wake-sleep curriculum procedure is the Library $\mathcal{L}$ built along the way (see Fig. 4.3). It incorporates knowledge of the domains encountered during the model's exposure to the curriculum, encoding the most useful concepts (functions) found for solving the tasks. From an algorithmic reasoning perspective, these are the best explanations provided by the model to the observations it was handed. However, we can rationalise the algorithms only if we understand the primitives, which may be true if alignment with the curriculum functions is enforced, but it is not guaranteed in general. For example, a learned concept could be a large network performing a complex task such as semantic segmentation or image classification, where the underlying algorithm is effectively a black box. Thus the algorithm produced is only interpretable to the extent that its components are understood.

4.2.1 Neural Primitives and Abstraction

The library could in principle include any neural network module, whether recurrent or convolutional, 7 parameters or 86 million. However, for the experiments below we initialise with a minimal number of relatively narrow hidden layers for new primitives in the *Primitive Inception* step (parameters found in Table A.1) as the tasks examined are simple enough for a handful of neurons to approximate well. As discussed in Section 3.2.2, these units can be assembled into *abstractions*, each a modular neural network itself, added to $\mathcal{L}$ for potential later reuse – this is illustrated in Fig. 4.3.

This setup means the new primitives will be optimised in a supervised manner, given input-output pairs from the task it needs to solve. The procedure for choosing when to train a new neural unit however, has to be designed with learnability in mind: if the examples contain more than one function, they may disagree on the output given the same inputs, or even be orthogonal to each other.

To minimise this effect, our method employs a relatively aggressive training strategy during primitive inception, coupled with a patient curriculum. In practice, this means high learning rate, low error tolerance threshold (specified in Table A.1) and a curriculum that only progresses if the primitive training has converged. Our findings regarding these considerations and empirical justification for them can be found in Section 5.3.

With the method defined, we move on to confirm the hypotheses in Section 1.1 and validate our approach in the following chapter.

Chapter 5

Evaluation

The following evaluation of our HBPS framework with neural concept libraries aims to demonstrate the model's capacity to find general programs given its learned primitives and to improve its library by creating new neural primitives. These experiments were designed to test the flexibility and robustness of the system with a focus on the role of curricula in its success. We explore the synthesis of simple arithmetic and boolean functions, as well as the individual performance of the phases of the wake-sleep cycle: wake, sleep (primitive inception, abstraction, and dreaming).

Our evaluation is based on tasks that require the synthesis of programs solving arithmetic and logic operations. These tasks were chosen for their simplicity, as well as their compositional nature, allowing a minimal demonstration of the utility of neural concept libraries and the hierarchical Bayesian framework. Additionally, by incorporating a curriculum of progressively complex tasks, we examine how structured learning influences the system's ability to generalise and synthesise programs.

5.1 Experimental Setup

We generate synthetic datasets to facilitate controlled experimentation and ease of evaluation. Each dataset consists of examples corresponding to a specific task, where functions are selected from a predefined set of Boolean and arithmetic operations (See Table 5.1). For each task, input values are randomly generated, and the corresponding output is computed by executing the source code (the target sequence). This process ensures a diverse range of input-output pairs for training and evaluating the system, but we stress that a representative set of examples is essential for good generalisation properties (e.g. the proportion of zeros as inputs to a Boolean problem).

The main metric we will examine is **function-matching accuracy**, measuring the alignment between generated proposals and target source code; ie. the proportion of correctly guessed functions. For a match to occur, the function identifier (e.g. `f12`) has to be correct and in the same location of the code when comparing the λ-expressions of the proposal and target. For example, `(f0 (f1 i0) f2)` would not be a fully correct prediction for `(f0 (f2 i0) f1)`, but it would gain a partial score of 1/3. This score is then averaged over the evaluation set, noting that it is normalised with respect to the total number of function applications, and not number of tasks. Values expressed as percentages include uncertainty estimates in the form of 95% confidence intervals.

Table 5.1 Categories of primitive operations in the full curriculum.

Boolean Logic	**Comparisons**	**Arithmetic**
AND	GreaterThan	Addition
OR	LessThan	Subtraction
NOT	EqualTo	Multiplication
XOR	GreaterThanEqualTo	Division
NAND	LessThanEqualTo	
NOR	NotEqualTo	
XNOR		

5.1.1 Boolean Tasks

The Boolean synthesis tasks involve generating programs that replicate the behaviour of basic binary logic gates, such as `AND`, `OR`, `XOR`, and `NOT`. Composing such gates is the basis of computing, making them ideal candidates for testing our system's ability to learn reusable concepts.

For each gate, the model is provided with random pairs of binary input-output examples. For instance, in the AND gate task, the system receives pairs such as:

$$\begin{aligned} 0, 0 &\to 0 \\ 1, 1 &\to 1 \\ 0, 1 &\to 0 \\ 0, 1 &\to 1 \\ 0, 0 &\to 1 \\ &\vdots \end{aligned}$$

The goal is to synthesise programs that replicate the observed behaviour, matching known functions if possible, or training a primitive if not. In the example above, the desired output is:

```
v0 = f0 ( i0, i1 )
( v0 ) ;
```

as long as `f0` represents a neural network corresponding to the `AND` gate operation. If there is no such function in $\mathcal{L}$, a neural primitive is trained on the examples above until convergence. The system's ability to generalise beyond individual logic gates is tested by presenting it with compositions of gates (e.g., synthesising a circuit that

combines AND and OR gates). This evaluates the model's capacity to combine learned primitives to solve more complex tasks.

5.1.2 Arithmetic Functions

We also include elementary arithmetic functions like addition and multiplication in the training set, by providing single-digit input examples in the same manner as before:

$$\begin{aligned} 3, 5 &\to 8 \\ 9, 0 &\to 9 \\ 1, 1 &\to 2 \\ &\vdots \end{aligned}$$

for the addition task, where the target is similar code as for Booleans, but with the function corresponding to the neural concept of addition in $\mathcal{L}$. The primitives are not expected to generalise to multiple digits or magnitudes outside the training range, as this is a known weakness of feed-forward networks [19].

5.1.3 Mixture of Domains

To highlight the domain adaptation capability of the system, we introduce tasks that mix Boolean and arithmetic operations. For example, the system may need to synthesise a program that takes Boolean inputs, performs logic operations, and then applies an arithmetic function to the result. An example task might be:

$$\begin{aligned} 1, 1, 5, 3 &\to 8 \\ 0, 0, 6, 4 &\to 0 \\ 0, 1, 5, 0 &\to 0 \\ &\vdots \end{aligned}$$

as generated by `AND(1, 1) * (5 + 3)`, or written `(* (AND 1 1) (+ 5 3))` as a λ-term. In this case, the system must recognise the presence of the two domains, identify which functions to use and route the variables to the correct input positions. This tests whether the model has the fundamental ability to handle cross-domain tasks and compose functions from multiple categories.

5.2 Separating the Phases

5.2.1 Wake Phase

In the wake phase, the system generates candidate programs based on its current library of primitives and neural concepts. We evaluate this stage of the cycle by using a previously trained model to generate proposals for a task, and compare them to the true target source code. Note that in general there are many programs that could give the same outputs to a finite set of inputs, which makes this metric uninformative if the number of function applications in the programs is large enough. In those cases, we can only use the weaker signal of execution loss, which compares the outputs of the generated program against the target source code for a range of inputs.

Below, we provide two examples to illustrate the program generation process: one showing the output of a model before it has fully converged on the correct solution, and the other demonstrating a successfully generated program after convergence. All examples were generated with the hyperparameter settings in Table A.2. Padding tokens are omitted for clarity.

Example generation (correct): In the following example, the system generates a valid and correct program that precisely matches the target source code for the task:

$$\begin{aligned} 1, 4, 0 &\to 4 \\ 3, 2, 1 &\to 7 \\ 2, 0, 2 &\to 2 \\ &\vdots \end{aligned} \quad \Rightarrow \quad \begin{aligned} &\texttt{v0 = f6 (i0 i1)} \\ &\texttt{v1 = f4 (v0 i2)} \\ &\texttt{(v1) ;} \end{aligned}$$

Here, the candidate program defines two variables, `v0` and `v1`, using the functions `f2` and `f1` respectively. The output is correctly structured, where `f1` operates on `v0`, and the final expression returns `v1`. This indicates that the model has learned an effective representation for the elements of the target program and input pairs, proposing the correct solution.

Example generation (before convergence): The following example shows a program generated for the same task, earlier in the training process, before the model has fully converged:

$$\begin{aligned} 1, 4, 0 &\to 4 \\ 3, 2, 1 &\to 7 \\ 2, 0, 2 &\to 2 \\ &\vdots \end{aligned} \quad \Rightarrow \quad \begin{aligned} &\texttt{v0 = f2 (i1 i0)} \\ &\texttt{v1 = f15 (v0 f1 v0)} \\ &\texttt{(v1) ;} \end{aligned}$$

In this case, the model has not yet learned the correct structure. The candidate program incorrectly applies `f15` instead of the desired `f1`, and attempts to pass additional arguments to `f15`, resulting in an invalid program. This illustrates the iterative nature of the training process during the wake phase, where the model gradually refines its understanding and program generation capabilities. While the structure is similar to the correct program, the erroneous application of `f15` reflects the system's intermediate state before convergence.

Comparing these two examples, we can see how the model's outputs evolve over time, moving from initially flawed programs to correctly synthesised solutions. As for the full wake-sleep cycle, in cases where the generated programs don't match the exact target, execution loss can still be used to evaluate the correctness of the behaviour by comparing the candidate program's outputs with those of the target.

Results for Boolean Tasks

In the context of Boolean tasks, the system relies heavily on the wake phase due to the binary simplicity and well-defined nature of Boolean logic. The system must generate accurate expressions for various tasks, such as constructing logic gates (`AND, OR, XOR`) or combining them into more complex circuits. As even large differences in program structure may still lead to the same truth table output, we observe overfitting to be a very common issue.

To illustrate this, consider the following input including every element of the truth table and a generated proposal:

$$\begin{array}{l} 0,0 \to 0 \\ 0,1 \to 0 \\ 1,0 \to 0 \\ 1,1 \to 1 \end{array} \quad \Rightarrow \quad \begin{array}{l} \texttt{v0 = f0 (i1 i0)} \\ \texttt{(v0) ;} \end{array}$$

where `f0` stands for the `AND` gate. While this generation completely fits the data presented, it is not the source code that generated these examples. In fact, the target in this case was:

```
v0 = NAND ( i1 i0 )
v1 = NOT ( v0 )
( v1 ) ;
```

It is, of course, not possible to guess the correct target in cases like this, and therefore it is unreasonable to expect the model to find it. However, if the synthesis is

trained past the first signs of convergence, we observe it does indeed always produce the true target. We attribute this to overfitting, whereby the model has memorised the exact inputs and their ordering, mapping them to this specific code target with certainty. To overcome this, we suggest early stopping during the Dreaming phase, or to curate a dataset with no such overlaps, pruning for unique truth tables.

Mixture Tasks and Depth

We perform additional testing on the system's adaptation to functional complexity, in the form of code *depth*. This measures the number of function application statements in a program – in source code form, this corresponds exactly to *number of lines* -1 (subtracting the last line, the return statement where functions are disallowed). For example, our previous example showcased a depth-mismatch where the proposal depth was one, while the target was two. From the example, we might speculate that depth seen during training affects the model's likelihood of generating programs of the same complexity.

To quantify this relationship, we train synthesis models provided with tasks generated by programs of varying depth, namely 1, 2, 3 and a uniform mixture of these. These trained models are then each evaluated on test sets containing problems of each one of depth 1, 2 and 3 separately. The results, shown in Table 5.2 in terms of function matching accuracy indicate that the model is significantly influenced by the depth of training examples, exhibiting a form of specialisation to the depths seen. While overall, models performed best on the depth 1 test set, followed by their own train depth, we also note that the mixture train set seems to retain a good balance of each complexity. This justifies our use of mixed depths during the full curriculum experiments.

Table 5.2 Function-matching performance of different train depths across evaluation depths, averaged over 50 random restarts. Maximal values by row highlighted.

Test Depth	**Training Depths**			
	1	**2**	**3**	**1,2 and 3**
1	**58.23 ± 2.5%**	26.01 ± 1.8%	24.02 ± 2.0%	47.72 ± 2.3%
2	7.47 ± 1.2%	**23.46 ± 1.5%**	12.40 ± 1.0%	17.69 ± 1.4%
3	4.92 ± 1.0%	8.08 ± 0.9%	**15.29 ± 1.3%**	6.64 ± 0.8%

5.2.2 Sleep Phase

Primitive Inception

In the simple domains tested, the success of the *Primitive Inception* stage relies solely on the quality of the training data in the given cycle. If there is contamination by means of leftover examples from another task that was failed to be accounted for by another program, the primitive will be a poor approximation of the intended learning outcome.

However, with isolated training of the different primitive tasks until convergence, we can achieve arbitrarily low loss on the tasks in Table 5.1. While it was hypothesised that for a compositional task it should be more effective to synthesise a program capturing that structure, this was not consistently observed to be the case. The failure to propose a program that achieved a loss below the threshold automatically results in a new primitive being created, even if the composition was a simple one.

5.3 Curriculum Learning

In this section, we evaluate how the system performs under a structured curriculum, where tasks are presented in increasing complexity. This is in contrast to mixed difficulties and function types presented all at once to the synthesis model. We observe significant improvements in the accuracy of function matching in the wake phase – up to 75%, an improvement of more than 15% (see Fig. 5.1). This may be due to the enforced alignment of learned primitives with their target functions, thanks to the curriculum not progressing to the next step until the intended outcome (e.g. learning the new function) is achieved.

5.3.1 Adding Primitives

We look at the same two domains from before, Boolean and arithmetic; in particular the success of the Primitive Inception step adding new neural concepts, is investigated.

Logic Gates

The most successful curricula began with one single-gate task at a time (e.g., `AND`, `OR`, `NOT`), followed by combinations of gates (e.g., `NAND` circuits) only after the primitives were successfully trained. This behaviour clearly highlights the fact that when there is

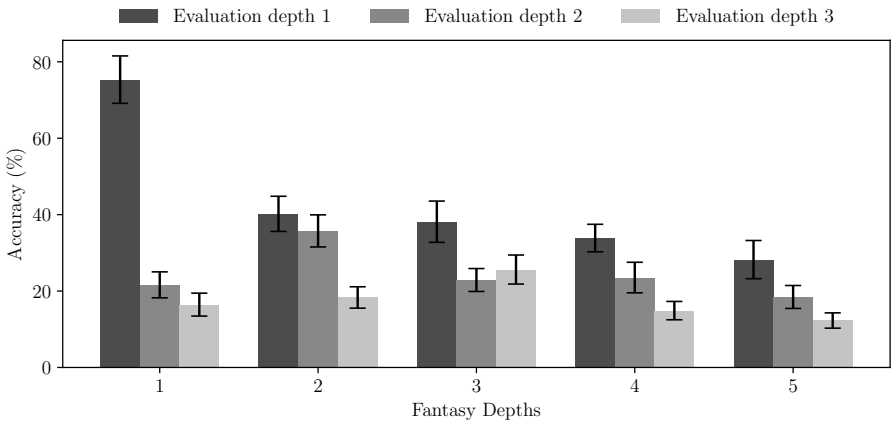

Fig. 5.1 Function matching accuracy evaluated with depths 1-3, varying the maximum fantasy generation depth in the Dreaming phase. Mean accuracies obtained from 5 different models each evaluated 20 times with random restarts.

a mixture of tasks present during *Primitive Inception* this hinders the ability of the model to optimise for a new primitive.

Arithmetic

Similar to the Boolean dataset, the arithmetic functions domain also benefited from isolated training of each concept through Primitive Inception. The compositions (e.g., an example like (+ i_1 (* i_0 i_1))) are only identified successfully by the synthesis model if the preceding cycles were successful, even the rounds for functions not included in the current composition. We suspect this may be due to more stability in the attention mechanism's training, as the data is cleaner if all neural concepts were are aligned correctly with their true generative counterparts.

5.3.2 Increasing Depth

We test the effect of fantasy depth on function matching accuracy, to investigate the sensitivity to this hyperparameter. As the results in Fig. 5.1. show, there is a clear tendency for the accuracy to peak when evaluation and fantasy depth are the same, with the highest score for the simple function applications (depth 1) at $75.34 \pm 6.2\%$. During training, we also observe that the allowed depth of the generated fantasies strongly correlates with performance on curricula with progressively increasing depth.

Chapter 6

Conclusions

6.1 Summary of Contributions

We have presented several contributions in the area of program synthesis, bringing together a number of perspectives, namely algorithmic reasoning, library learning and HBPS. Starting from a Bayesian inference framework most similar to that of DreamCoder, we introduced a novel extension of its wake-sleep cycle, the Primitive Inception phase (Section 4.1.3), where a new neural primitive is trained on tasks that could not be explained by the synthesis model. This approach enables systems to develop reusable, modular internal representations in the form of neural programs, with the potential for greater flexibility and adaptability across a range of tasks than previous approaches with manually designed DSLs.

A second contribution is the application of the encoder-decoder transformer architecture of [62] for translating from input-output pairs to the source code functional programming language (Section 4.1.2). This is a key part of our implementation that allows the learning of a hierarchical inference scheme encoding the grammar, as well as enabling length-generalisation with respect to inputs and generations via the attention mechanism used in predicting the tokens one at a time.

Additionally, we highlight the role of our design and implementation of a dynamic curriculum learning framework that gradually increases task complexity, enabling more efficient learning of basic primitives at Inception and better embeddings for these during Dreaming. By focusing on modularity and reusability, this training framework improves the system's ability to learn more robust representations that are more likely to generalise across diverse tasks.

Lastly, we conducted an evaluation as a proof-of-concept to demonstrate the effectiveness of our approach across various domains, including Boolean, arithmetic, and simple compositional algorithmic reasoning tasks (Section 5.2.1). Our experiments confirm that the system is able to obtain new knowledge and incorporate it in its concept library for reuse, purely through the examples it is provided with in the curriculum.

We claim that these contributions collectively advance the field by presenting a method of program synthesis in a neural functional programming language continually developed by the model purely from experience. The suggested framework with

curriculum learning presents a pathway toward systems that can autonomously develop and refine functional abstractions across a variety of contexts, laying the groundwork for future research in more general, domain-agnostic program synthesis and algorithmic reasoning.

6.2 Limitations

Despite the strengths of the proposed approach, several limitations should be acknowledged. First, while the method performs well on smaller, modular tasks, it has not been confirmed to be able to scale as the complexity of the program space grows. Larger or recursive programs still pose a challenge for the system, as the current method does not provide the necessary tools for this type of generalisation.

Another limitation concerns our trading of reliance on predefined primitives for the necessity of a well-curated training curriculum instead. The system's performance is highly sensitive to the pace and progression of the learning tasks and the set of primitives it tries to mimic. In cases where the example set is not adequately representative of a program's behaviour, or the curriculum is poorly designed, the system will likely struggle to form effective abstractions and perform poorly on more complex tasks. In other words, this approach with its adoption of purely neural primitives does not eliminate the need for human domain knowledge, but shifts its role from manually designed primitives to carefully crafted curricula to ensure quality and representativeness.

On a more technical note, the execution loss metric used for evaluating program correctness in certain cases, especially when exact match criteria are not feasible, can lead to misleading conclusions. Programs that behave similarly in terms of output but are structurally different from the target program have no way of surfacing, even though they may be preferable in terms of simplicity. This can obscure the system's ability to learn the correct underlying representations, highlighting the need for more sophisticated datasets as well as revised fantasy generation in the Dreaming phase (Section 4.1.3).

Finally, a significant limitation of this work was the constraint on both time and computational resources. With just three months allocated for the project and limited access to high-performance GPUs, the scope of experiments was necessarily constrained. This meant that certain experiments had to be prioritised over others, leaving limited room for deeper exploration or richer datasets, or to conduct extensive hyperparameter tuning. As a result, certain aspects of the system, such as the full scalability of the

approach and its application to more complex or deeper tasks, could not exhaustively be tested. These resource constraints also limited the breadth of tasks that could be evaluated, and the performance could likely be further improved with access to more compute for larger-scale experiments.

6.3 Further Work

We believe there are several promising directions for future research building on the work presented. First, improving the system's scalability is critical, particularly in handling more complex, recursive tasks. We suggest two ways of tackling this: incorporating recurrent neural networks (RNNs) [66] as modules to handle sequential tasks; or leaving it to the function compiler (Section 4.1.1) and abstractions to identify and incorporate `map`, `fold` and `filter`.

Similarly, every domain has its own set of inductive biases, along with specialised architectures to capture these, such as convolutions in vision or transformers in language processing. For a truly generalisable primitive acquiry process, we cannot rely on feed-forward networks only. Constructing neural blocks with more architectural freedom in connections and structure would certainly allow broader generalisation capabilities.

Additionally, integrating stronger mechanisms for handling contradictory data would be beneficial. To minimise the system's sensitivity to the quality of the curriculum, it needs to be robust against imperfections in the data provided. Developing a mechanism to identify which examples fit into a task (by e.g. a learned similarity measure) could significantly enhance the practical applicability of the approach.

From a computational theory perspective, more sophisticated memory management could be employed for better efficiency, while adoption of fuzzy type systems [67] would aid search and robustness.

Finally, we propose performing a set of experiments, particularly in a perceptional-programming-by-example (PPBE) context, to further showcase the advantages of using neural primitives trained on-the-fly, while adhering to a strict formal grammar. For example the datasets used in demonstrating DreamCoder's abilities in abstraction, or a perceptual task coupled with algorithmic elements (such as those presented in [35, 37]).

While challenges remain, particularly in scalability and robustness, the potential for broad applicability across diverse domains is encouraging. As research in this field progresses, we anticipate the development of systems capable of handling more complex, generalisable tasks with minimal human oversight, paving the way for more adaptive, reliable intelligent machines.

Acknowledgements

I would like to express my gratitude to my project supervisors for providing me with valuable guidance and support throughout this summer. Their insights have not only shaped the direction of this project but also influenced my broader approach to problem-solving and critical thinking.

I am deeply grateful to my family for their constant support and for the invaluable opportunities they have provided me.

Finally, I extend my thanks to my partner for her unwavering encouragement and engaging discussions about various related topics.

References

[1] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. Place: US Publisher: American Psychological Association.

[2] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, April 1982.

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

[4] Yutaka Matsuo, Yann LeCun, Maneesh Sahani, Doina Precup, David Silver, Masashi Sugiyama, Eiji Uchibe, and Jun Morimoto. Deep learning, reinforcement learning, and world models. *Neural Networks*, 152:267–275, 2022.

[5] A. M. Turing. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, October 1950.

[6] Daniel C. Dennett. Intentional Systems. *Journal of Philosophy*, 68(February):87–106, 1971. Publisher: Bradford Books.

[7] Gualtiero Piccinini and Sonya Bahar. Neural Computation and the Computational Theory of Cognition. *Cognitive Science*, 37(3):453–488, 2013. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cogs.12012.

[8] Michael Rescorla. The Computational Theory of Mind. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2020 edition, 2020.

[9] Jerry A. Fodor. *The Language of Thought*. Harvard University Press, 1975.

[10] Aarohi Srivastava et al. Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models, June 2023. arXiv:2206.04615 [cs, stat].

[11] Yi Ru Wang, Jiafei Duan, Dieter Fox, and Siddhartha Srinivasa. NEWTON: Are Large Language Models Capable of Physical Reasoning?, October 2023. arXiv:2310.07018 [cs].

[12] Yuxuan Wan, Wenxuan Wang, Yiliu Yang, Youliang Yuan, Jen-tse Huang, Pinjia He, Wenxiang Jiao, and Michael R. Lyu. A & B == B & A: Triggering Logical Reasoning Failures in Large Language Models, January 2024. arXiv:2401.00757 [cs].

[13] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: An Adversarial Winograd Schema Challenge at Scale, November 2019. arXiv:1907.10641 [cs].

[14] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. CommonsenseQA: A Question Answering Challenge Targeting Commonsense Knowledge, March 2019. arXiv:1811.00937 [cs].

[15] Mikel Bober-Irizar and Soumya Banerjee. Neural networks for abstraction and reasoning: Towards broad generalization in machines, February 2024. arXiv:2402.03507 [cs].

[16] François Chollet. On the Measure of Intelligence, November 2019. arXiv:1911.01547 [cs].

[17] Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, Roman Wang, Nikhil Singh, Taylor L. Patti, Jayson Lynch, Avi Shporer, Nakul Verma, Eugene Wu, and Gilbert Strang. A Neural Network Solves, Explains, and Generates University Math Problems by Program Synthesis and Few-Shot Learning at Human Level. *Proceedings of the National Academy of Sciences*, 119(32):e2123433119, August 2022. arXiv:2112.15594 [cs].

[18] Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, January 2024.

[19] Alberto Testolin. Can Neural Networks Do Arithmetic? A Survey on the Elementary Numerical Skills of State-of-the-Art Deep Learning Models. *Applied Sciences*, 14(2):744, January 2024.

[20] Tarek R. Besold, Artur d'Avila Garcez, Sebastian Bader, Howard Bowman, Pedro Domingos, Pascal Hitzler, Kai-Uwe Kuehnberger, Luis C. Lamb, Daniel Lowd, Priscila Machado Vieira Lima, Leo de Penning, Gadi Pinkas, Hoifung Poon, and Gerson Zaverucha. Neural-Symbolic Learning and Reasoning: A Survey and Interpretation, November 2017. arXiv:1711.03902 [cs].

[21] Irina Higgins, Nicolas Sonnerat, Loic Matthey, Arka Pal, Christopher P. Burgess, Matko Bosnjak, Murray Shanahan, Matthew Botvinick, Demis Hassabis, and Alexander Lerchner. SCAN: Learning Hierarchical Compositional Visual Concepts, June 2018. arXiv:1707.03389 [cs, stat].

[22] Artur d'Avila Garcez and Luis C. Lamb. Neurosymbolic AI: The 3rd Wave, December 2020. arXiv:2012.05876 [cs].

[23] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, October 2016.

[24] Kensen Shi, Hanjun Dai, Wen-Ding Li, Kevin Ellis, and Charles Sutton. LambdaBeam: Neural Program Search with Higher-Order Functions and Lambdas, October 2023. arXiv:2306.02049 [cs].

[25] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, March 1971.

[26] Percy Liang, Michael Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. pages 639–646, 08 2010.

[27] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning, June 2020. arXiv:2006.08381 [cs].

[28] Geoffrey E. Hinton, Peter Dayan, Brendan J. Frey, and Radford M. Neal. The "Wake-Sleep" Algorithm for Unsupervised Neural Networks. *Science*, 268(5214):1158–1161, May 1995.

[29] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 317–330, Austin Texas USA, January 2011. ACM.

[30] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages, September 2014. arXiv:1409.2378 [cs].

[31] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural Programmer: Inducing Latent Programs with Gradient Descent, August 2016. arXiv:1511.04834 [cs, stat].

[32] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable Programs with Neural Libraries, March 2017. arXiv:1611.02109 [cs].

[33] Andrew Trask, Felix Hill, Scott Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural Arithmetic Logic Units, August 2018. arXiv:1808.00508 [cs].

[34] Andreas Madsen and Alexander Rosenberg Johansen. Neural Arithmetic Units. January 2020. arXiv:2001.05016 [cs].

[35] Scott Reed and Nando de Freitas. Neural Programmer-Interpreters. February 2016. arXiv:1511.06279 [cs].

[36] Noah D. Goodman, Joshua B. Tenenbaum, Thomas L. Griffiths, and Jacob Feldman. Compositionality in Rational Analysis: Grammar-Based Induction for Concept Learning. In Nick Chater and Mike Oaksford, editors, *The Probabilistic Mind: Prospects for Bayesian Cognitive Science*. Oxford University Press, 2008.

[37] Maxwell I. Nye, Armando Solar-Lezama, Joshua B. Tenenbaum, and Brenden M. Lake. Learning Compositional Rules via Neural Program Synthesis, October 2020. arXiv:2003.05562 [cs].

[38] Jerome S. Bruner, Jacqueline J. Goodnow, and George A. Austin. *A study of thinking.* A study of thinking. John Wiley and Sons, Oxford, England, 1956. Pages: xi, 330.

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[40] John McCarthy. Programs with common sense. Technical report, USA, 1960.

[41] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What Can Neural Networks Reason About? 2019. Publisher: [object Object] Version Number: 4.

[42] Linghan Zhong, Ryan Lindeborg, Jesse Zhang, Joseph J. Lim, and Shao-Hua Sun. Hierarchical Neural Program Synthesis, March 2023. arXiv:2303.06018 [cs].

[43] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural Program Learning under Noisy I/O, March 2017. arXiv:1703.07469 [cs].

[44] Garrett E. Katz, Khushboo Gupta, and James A. Reggia. Reinforcement-based program induction in a neural virtual machine. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2020.

[45] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models, October 2023. arXiv:2305.16291 [cs].

[46] Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical Programmatic Reinforcement Learning via Learning to Compose Programs. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 21672–21697. PMLR, July 2023.

[47] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural Architecture Search: Insights from 1000 Papers, January 2023. arXiv:2301.08727 [cs, stat].

[48] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical Representations for Efficient Architecture Search, February 2018. arXiv:1711.00436 [cs, stat].

[49] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-Down Synthesis for Library Learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, January 2023. arXiv:2211.16605 [cs].

[50] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines, December 2014. arXiv:1410.5401 [cs].

[51] Petar Veličković and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2(7):100273, 2021.

[52] Vladimir V. Mirjanić, Razvan Pascanu, and Petar Veličković. Latent Space Representations of Neural Algorithmic Reasoners, July 2023. arXiv:2307.08874 [cs, stat].

[53] Wilfried Bounsi, Borja Ibarz, Andrew Dudzik, Jessica B. Hamrick, Larisa Markeeva, Alex Vitvitskyi, Razvan Pascanu, and Petar Veličković. Transformers meet Neural Algorithmic Reasoners, June 2024. arXiv:2406.09308 [cs].

[54] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, October 2018. arXiv:1806.01261 [cs, stat].

[55] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural Execution of Graph Algorithms, January 2020. arXiv:1910.10593 [cs, stat].

[56] Alonzo Church. The calculi of lambda-conversion. 1941.

[57] Janis Zenkner, Lukas Dierkes, Tobias Sesterhenn, and Chrisitan Bartelt. Abstract-Beam: Enhancing Bottom-Up Program Synthesis using Library Learning, July 2024. arXiv:2405.17514 [cs].

[58] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[59] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Joshua B. Tenenbaum. Library learning for neurally-guided Bayesian program induction. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 7816–7826, Red Hook, NY, USA, 2018. Curran Associates Inc. event-place: Montréal, Canada.

[60] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, page 1302–1309. AAAI Press, 2013.

[61] Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is All You Need: Learning Skills without a Reward Function, October 2018. arXiv:1802.06070 [cs].

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. arXiv:1706.03762 [cs].

[63] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 41–48, Montreal Quebec Canada, June 2009. ACM.

[64] Mary Phuong and Marcus Hutter. Formal Algorithms for Transformers, July 2022. arXiv:2207.09238 [cs].

[65] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[66] M I Jordan. Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986. 5 1986.

[67] Nasim Rahaman, Muhammad Waleed Gondal, Shruti Joshi, Peter Gehler, Yoshua Bengio, Francesco Locatello, and Bernhard Schölkopf. Dynamic Inference with Neural Interpreters, October 2021. arXiv:2110.06399 [cs].

Appendix A

Hyperparameter Settings

In this appendix, we provide the hyperparameter settings used in the various phases of the wake-sleep cycle and the transformer model training. These are the settings required for replicating the experimental results discussed in Sections 5.3 and 5.2.1.

A.1 Wake-Sleep Hyperparameter Settings

Table A.1 lists the hyperparameters used during the wake-sleep phases, including the Waking, Primitive Inception, and Dreaming phases. Each phase requires its own set of parameters to guide the training process in a stable manner.

Table A.1 Hyperparameter settings for different phases of wake-sleep.

Hyperparameter	**Value**
Waking:	
Number of Tasks	1000
Number of Examples per Task	16
Number of Proposals	10
Primitive Inception:	
Initial Concept Width	2
Initial Concept Height	1
Threshold Acceptance Loss	10^{-2}
Inception Learning Rate	10^{-2}
Inception Epochs	1000
Dreaming:	
Number of Fantasies	10^5
Number of Fantasy Examples	8
Fantasy Depth	3
Dreaming Learning Rate	10^{-4}
Dreaming Epochs	500

A.2 Transformer Hyperparameter Settings

Table A.2 presents the hyperparameters used in the transformer-based synthesis model training (see Section 5.2.1). These settings determine the architecture of the transformer, including the number of attention heads, and embedding size, which the method is sensitive to in terms of performance.

Table A.2 Transformer hyperparameter settings used.

Hyperparameter	**Value**
Number of Attention Heads	8
Head Size	32
Embedding Size	512
Encoder-Decoder Layers	4
Feed-forward Hidden Dimension	256
Context Window Length	64