Bayesian Program Learning by Decompiling Amortized Knowledge

Alessandro B. Palmarini [1 †] **Christopher G. Lucas** [2] **N. Siddharth** [2 3]

Abstract

DREAMCODER is an inductive program synthesis system that, whilst solving problems, learns to simplify search in an iterative wake-sleep procedure. The cost of search is amortized by training a neural search policy, reducing search breadth and effectively "compiling" useful information to compose program solutions across tasks. Additionally, a library of program components is learnt to compress and express discovered solutions in fewer components, reducing search depth. We present a novel approach for library learning that directly leverages the neural search policy, effectively "decompiling" its amortized knowledge to extract relevant program components. This provides stronger amortized inference: the amortized knowledge learnt to reduce search breadth is now also used to reduce search depth. We integrate our approach with DREAMCODER and demonstrate faster domain proficiency with improved generalization on a range of domains, particularly when fewer example solutions are available.

1. Introduction

The goal in inductive program synthesis is to generate a program whose functionality matches example behaviour (Gulwani et al., 2017). If behaviour is specified as input/output examples, then the problem could be solved trivially by defining a program that embeds (hard codes) the example transformations directly. This solution, however, does not provide an account relating, or explaining, how inputs map to outputs, nor does it facilitate generalization to inputs beyond what it has observed. Discovering a program that can do this cannot be derived from examples, just as our scientific theories cannot be derived from observations (Deutsch, 2011)—some form of hypothesising is required (i.e. search).

[†]Majority of work done at University of Edinburgh [1]Santa Fe Institute, Santa Fe, NM, USA [2]School of Informatics, University of Edinburgh, Edinburgh, UK [3]The Alan Turing Institute, UK. Correspondence to: Alessandro B. Palmarini <abp@santafe.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

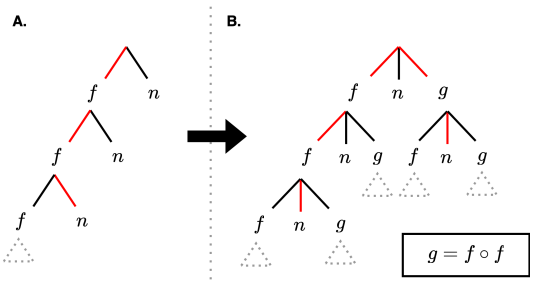

Figure 1: **(A)** Program search space for a simple library containing two primitives: a function $f : \mathbb{N} \to \mathbb{N}$ and a variable $n : \mathbb{N}$. Red path highlights the program $f(f(n))$. **(B)** Restructured search space after $f \circ f$ is chunked into a new primitive g, illustrating the breadth-depth trade off.

Any program must be built from some set of primitive components: base operations and elements. While a set of such components (library) is finite, they can be combined in an infinite number of ways to form different programs. Moreover, this number grows exponentially with the number of components used, rendering blind search intractable in all but the simplest of cases. Improving upon blind search requires reducing the number of combinations searched. There are two broad routes to achieving this: we can (i) reduce the *breadth* searched as we move down the search tree to program solutions or we can (ii) reduce the *depth* searched by pushing program solutions higher up the search tree. Accomplishing either requires additional *knowledge*.

To reduce the breadth searched requires knowledge for avoiding search paths that do not contain program solutions, while focusing on those that do—i.e. a search policy. To reduce the depth searched requires knowledge about which functionality is used by program solutions: if any functionality, expressed as a combination of library components, was itself made part of the library, then any program using this functionality could be expressed with fewer components. In effect, a copy of every program containing the functionality is pasted higher up the search tree, reducing the depth required to reach the program's corresponding computation, as illustrated in Fig. 1. We refer to turning composed functionality into a library operation as "chunking".

Chunking to reduce the search depth comes at the cost of increasing the search breadth—there are now more library

components to consider (Fig. 1). Hence, *the knowledge for successfully reducing the depth searched in a program induction task is highly reliant on the knowledge used to reduce the breadth*, and vice versa. This interplay is key to the main problem addressed in this paper.

Practical and successful program synthesis approaches, such as *FlashFill* (Gulwani, 2011), involve the careful selection of its library components, i.e., the domain-specific language (DSL) used. Human experts define a search space where programs capable of solving specific tasks can be found tractably, but these systems are typically unable to solve tasks not anticipated by the designers—failing to generalize. A general-purpose inductive program synthesis system, then, cannot rely on handcrafted DSLs; instead, it must learn how to structure its search space, via the components contained in its library (Dechter et al., 2013; Dumancic et al., 2021; Shin et al., 2019; Iyer et al., 2019), and learn how to navigate that search space for program solutions (Balog et al., 2016; Devlin et al., 2017; Kalyan et al., 2018; Chen et al., 2018).

DreamCoder (Ellis et al., 2021; 2023) is an inductive program synthesis system that jointly learns a library and a probabilistic search policy, but without explicitly considering the interplay between search depth and breadth described above. That is, the knowledge learnt to guide the search space has no direct effect on how it is restructured (i.e., how the library is learnt). We contribute a novel approach for library learning that directly leverages the existing knowledge learnt to guide search, enabling the extraction of more useful, and complementary, components from the same experience. We integrate our approach with DreamCoder and demonstrate faster domain proficiency with improved generalization on a range of program synthesis domains.

2. Background

We begin with an overview of DreamCoder's approach: viewing program induction as probabilistic inference (Lake et al., 2015). Consider a set of program induction tasks $\mathcal{X}$. We assume each task $x \in X$ was produced by an unobserved (latent) program ρ and each ρ was generated by some prior distribution $p_{\boldsymbol{\theta}^*_{\mathcal{D}}}(\rho)$. We assume the prior is defined by model parameters $\boldsymbol{\theta}_{\mathcal{D}}$ that specify the probability that components part of a library $\mathcal{D}$ are used when generating programs (Liang et al., 2010; Menon et al., 2013). The true library and component parameters $\boldsymbol{\theta}^*_{\mathcal{D}}$ are unknown.

The marginal likelihood of the observed tasks is then given by $p_{\boldsymbol{\theta}_{\mathcal{D}}}(\mathcal{X}) = \prod_{x \in \mathcal{X}} \sum_{\rho} p(x \mid \rho) p_{\boldsymbol{\theta}_{\mathcal{D}}}(\rho)$, where $p(x \mid \rho)$ is the likelihood of x being produced, and hence solved, by ρ. To learn a good generative model that maximises the likelihood, we need to know which programs score highly under $p(x \mid \rho)$—i.e. solve our tasks. We have seen previously why this is challenging: discovering programs that can account for the observed tasks requires search.

To help with search, a recognition (inference) model $q_{\boldsymbol{\phi}_{\mathcal{D}}}(\rho \mid x)$ is learnt to infer the programs that are most likely to solve a given task. The recognition model parameters $\boldsymbol{\phi}_{\mathcal{D}}$ map tasks to distributions over programs that, as with the generative model, specifies the probability that components part of the library $\mathcal{D}$ are used. Estimating $\boldsymbol{\phi}_{\mathcal{D}}$ is done using both (ρ, x) pairs sampled from the generative model (fantasies) and programs found to solve the observed tasks $x \in \mathcal{X}$ (replays). The recognition model is used to search for programs that solve a task x by enumerating programs in decreasing order of their probability under $q_{\boldsymbol{\phi}_{\mathcal{D}}}(\rho \mid x)$. Programs that solve x are stored in a task-specific set $\mathcal{B}_x$.

Discovering programs that may have produced the observed tasks (those in $\{\mathcal{B}_x\}_{x \in \mathcal{X}}$) now provides more data to infer the parameters $\boldsymbol{\theta}_{\mathcal{D}}$ generating them. Inferring $\boldsymbol{\theta}_{\mathcal{D}}$ entails choosing the library $\mathcal{D}$ whose components they control. Rather than maximise the likelihood directly, DreamCoder performs maximum a posteriori (MAP) inference using a prior over libraries $\mathcal{D}$ and parameters $\boldsymbol{\theta}_{\mathcal{D}}$. Maximising the MAP objective (which can only be approximated) w.r.t. $\mathcal{D}$ corresponds to updating $\mathcal{D}$ to include functions that best compress the discovered solutions. After updating $\mathcal{D}$, parameters $\boldsymbol{\theta}_{\mathcal{D}}$ are updated to their MAP estimates.

This completes a single learning cycle with DreamCoder—a variant of the wake-sleep algorithm (Hinton et al., 1995; Bornschein & Bengio, 2014; Le et al., 2020). An iterative procedure is used to jointly learn a generative and inference model (Dayan et al., 1995), while additionally searching for program solutions. After updating the generative model, it can then be used to sample (ρ, x) pairs more indicative of those coming from the true generating process. These can be used to learn a more accurate recognition model with improved inference (dream sleep), which results in more programs capable of solving tasks discovered during search (wake). More program solutions lead to better estimates of the generative model (abstraction sleep). Each stage bootstraps off that learnt in the previous stage.

3. Dream Decompiling

Learning a library $\mathcal{D}$ and recognition model parameters $\boldsymbol{\phi}_{\mathcal{D}}$ during DreamCoder's two sleep phases corresponds to learning the two types of knowledge (discussed in Section 1) to simplify program search (Fig. 2 black arrows). Adding functions to $\mathcal{D}$ reduces the search depth to reach programs utilizing those functions. Searching for programs under $q_{\boldsymbol{\phi}_{\mathcal{D}}}(\rho \mid x)$ directs the search down preferred branches. Chunking (to reduce search depth) restructures the search space that the recognition model guides (to reduce search breadth). How the search space is restructured (here by adding to $\mathcal{D}$) should complement the search policy (here

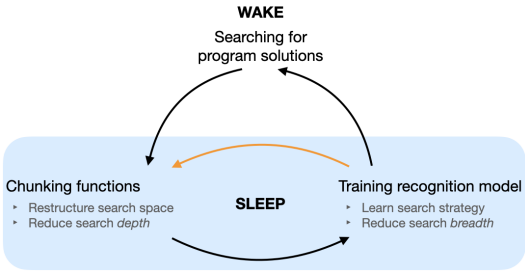

Figure 2: Positive feedback loop between searching for program solutions and learning to simplify search in DREAMCODER (black arrows). In dream decompiling (orange arrow) the knowledge learnt to reduce search breath directly influences the knowledge learnt to reduce search depth.

$q_{\phi_{\mathcal{D}}}(\rho \mid x)$) guiding that space. For example, consider a task with multiple solutions: reducing the depth to reach some of those solutions does not simplify search if the guide is focused on others. In DREAMCODER, the recognition model has only an indirect effect on what is chunked through the program solutions it helps discover (as they then determine which functions are most compressive). This raises the question: can we improve inference by *directly* leveraging the knowledge learnt by the recognition model to decide what is chunked, and hence have the recognition model shape the search space that it will subsequently learn to guide?

The recognition model uses one set of parameters ($\phi_{\mathcal{D}}$) to model the relation between observed tasks and latent programs that may have produced those tasks. A single function (parameterised by $\phi_{\mathcal{D}}$) is used to specify an inference distribution $q(\rho \mid x)$ for any x, rather than learning each individually. This is called amortized inference (Gershman & Goodman, 2014; Stuhlmüller et al., 2013): search on new tasks becomes tractable by reusing knowledge learnt from past inferences. By leveraging the recognition model to decide what to chunk, we now amortize the cost of search (inference) in two ways: recognition model parameters $\phi_{\mathcal{D}}$ are reused to reduce both search breadth *and* depth.

The amortized model learnt by the recognition model offers the potential to chunk valuable functions for *unsolved* tasks, due to its ability to generalize. Unsolved tasks have no example solutions to infer compressive functionality. Moreover, the functionality crucial for solving new tasks may not be present or sufficiently abundant in existing solutions to other tasks. Such functionality is thus overlooked by compression-focused approaches to library learning: dealing with future uncertainty is often antagonistic to optimal compression of what has worked in the past (Chollet, 2019).

Training the recognition model on (ρ, x) pairs "compiles" useful information into model parameters $\phi_{\mathcal{D}}$ for generating program solutions across tasks (Le et al., 2017). The fantasy (ρ, x) pairs sampled from the generative model are commonly referred to as "dreams" (Murphy, 2023). Chunking with the recognition model (Fig. 2 orange arrow) involves utilizing the compiled information to identify the most useful functions to incorporate into the library, essentially *decompiling* the knowledge acquired through amortized inference. This is analogous to a decompiler translating compiled machine code into high-level source code. Hence, we refer to this concept as "dream decompiling".

4. Approach

In this section we present two variants of dream decompiling, addressing the problem of choosing effective library components. For simplicity, we consider the case where the recognition model defines a probability distribution over library components and that the probability of generating a program is equal to the product of independently generating each component of the program. Appendix A describes how this extends to bigram distributions over well-typed programs (as is the case with DREAMCODER).

A simple approach to chunking with the recognition model is to consider which functions it wants to use most often, irrespective of the particular task being solved:

$$q_{\phi_{\mathcal{D}}}(f) = \mathbb{E}_{x \sim \mathcal{X}}[q_{\phi_{\mathcal{D}}}(f \mid x)] \tag{1}$$

We refer to this variant as DREAMDECOMPILER-AVG. While this provides a means to rank each f in a set of candidates $\mathcal{F}$, there are two problems with this approach:

i. A ranking can determine if chunking one function is better than another, but it provides no insight into how many functions (if any) should be chunked.
ii. Larger functions are naturally harder to generate than smaller functions and hence disfavoured in the ranking. To see why this can be problematic, consider a strict subfunction s of a function f. On all tasks, the recognition model is less likely to generate f than s as it would need to generate all components in s plus more. But if the recognition model is only ever intending to generate s as part of f, then chunking f is better: future use would require generating a single component instead of multiple and s has no use elsewhere.

If we want to leverage the recognition model for chunking, we need a criterion that (i) determines which functions should actually be chunked and (ii) considers the overall impact of chunking a function, not only generation preference.

It is unknown whether chunking a function will enhance, diminish, or have no impact on the recognition model's ability to guide the search for program solutions. A formal way to handle uncertain knowledge is with probability distributions (Cox, 1946). In this section we introduce a probabilistic model to systematically express the uncertainty

in whether or not chunking a given function will have an overall beneficial effect on the recognition model's future ability to guide search for program solutions. The model, written $p(c \mid f; \phi_{\mathcal{D}})$, is a Bernoulli distribution over the binary random variable c that, w.r.t. some function f, has the value 1 if the net effect of chunking f is positive and 0 otherwise. The distribution used depends on the recognition model $q_{\phi_{\mathcal{D}}}(\rho \mid x)$ and is hence parameterised by $\phi_{\mathcal{D}}$.

With the model $p(c \mid f; \phi_{\mathcal{D}})$ we can solve both of the above problems. The first can be solved using decision theory (DeGroot, 2005). For example, if the utility gained from a favorable chunk was equal to the cost incurred from an unfavorable chunk, then opting to chunk a function f only when $p(c \mid f; \phi_{\mathcal{D}}) \geq 0.5$ would maximise the expected payoff.[1] The extent to which the second problem is solved depends on how well the true distribution is modeled by $p(c \mid f; \phi_{\mathcal{D}})$, which we now explain how to do effectively.[2]

A model of $p(c \mid f; \phi_{\mathcal{D}})$ would require balancing the effect of chunking on the recognition model's future ability to generate all desired programs, across all tasks. The problem can be simplified by introducing and then marginalising out the programs and tasks instead:

$$p(c \mid f; \phi_{\mathcal{D}}) = \sum_{x} \sum_{\rho} p(c \mid f, \rho, x; \phi_{\mathcal{D}}) p(\rho, x \mid f; \phi_{\mathcal{D}}) \quad (2)$$

We will return to the double summation and joint $p(\rho, x \mid f; \phi_{\mathcal{D}})$ later. The problem is now reduced to needing a model, $p(c \mid f, \rho, x; \phi_{\mathcal{D}})$, expressing uncertainty in whether or not chunking f will have a positive effect on the recognition model's ability to generate ρ for solving x.

Note that knowing f is part of ρ (which can solve x) does not guarantee a positive effect from chunking f. After f is added to the library, the recognition model parameters $\phi_{\mathcal{D}}$ (dependent on $\mathcal{D}$) are updated, defining a new distribution over programs in terms of the new library. The effect of chunking f is a *balance* between the improvement gained in generating f and any costs incurred in generating the rest of ρ. We can not know the exact effect of chunking as we do not know how $\phi_{\mathcal{D}}$ will be updated.

A related effect that (i) does not depend on how $\phi_{\mathcal{D}}$ will be updated, (ii) can be captured precisely and (iii) calculated efficiently is how beneficial is it for the recognition model to generate ρ if it does not need to generate f at all—which we refer to as "caching" f. In terms of the balance mentioned above, this is equivalent to the optimistic view that chunking f will result in maximal improvement for generating f (it can be generated for "free"), while also not incurring any cost for generating the rest of ρ.

Desiderata for a measure of caching benefit Let us consider a program ρ generated by a distribution $q(\rho)$ via the composition of smaller library components. We are interested in a mathematical way to quantify the benefit gained, from 0 (not useful at all) to 1 (as useful as can be), in generating ρ if instead the function f was cached (no longer required to be sampled), which we denote $\mathbb{C}(q, \rho, f)$. Any such measure should satisfy the following properties:

D1: (*min*) If f is not utilized by ρ, then caching f provides no benefit and $\mathbb{C}(q, \rho, f)$ should be 0.

D2: (*max*) The largest benefit is gained from caching the entire program (no matter how q is defined). If $f = \rho$, then $\mathbb{C}(q, \rho, f)$ should be 1.

D3: (*monotonic—changing programs*) Consider a fixed function f used in different ρ. The less likely q is to generate the parts of ρ that are not part of f, the smaller $\mathbb{C}(q, \rho, f)$ should be because caching f has a smaller impact on helping ρ to be subsequently generated. As an extreme example, if the probability of generating the rest of our program became 0, then caching f does not help with then generating the full program at all.

D4: (*monotonic—changing functions*) Consider different functions f used within the same initial program. The smaller $q(f)$ is, the better it is to cache (not needing to generate) f and hence the larger $\mathbb{C}(q, \rho, f)$ should be.

The first two properties are straightforward to understand. The last two properties are illustrated further in Fig. 3.

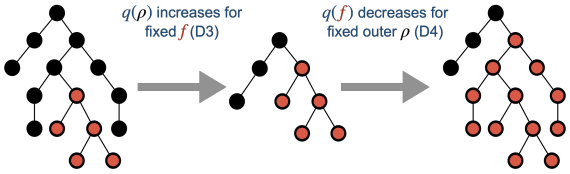

Figure 3: Program trees with functions (red subtrees) considered for caching. The probability of being generated by q decreases with an increase in nodes. Arrows indicate changes where caching becomes more beneficial. Caching the function in the first program is unhelpful, as the program remains difficult to generate. Caching the same function in the second program is more helpful, transitioning the program from relatively unlikely to likely. Caching the function in the third program is even more helpful: although post-cache sample probability is equal to the second, generating the third program was initially much more unlikely.

A direct consequence of the last two properties is that the more components of a program cached as part of the function, the more beneficial the caching is. The last two properties are two sides of the same coin: both suggest that the benefit from caching f is related to the *relative propor-*

[1] This is only true for symmetric payoffs. Exploring payoffs and their assignment is left for future work. Here, the simplest quantity selection suffices to showcase the strengths of the approach.

[2] Note that calculating the exact value of c with respect to all future tasks is not possible, and hence one can only have a heuristic.

tion of sample difficulty removed when generating ρ. In information-theoretic terms, "difficulty" can be used interchangeably with the "self-information" or "surprise" inherent in a random outcome (such as a function being generated) (MacKay, 2003). Capturing the self-information of a random outcome formally can be done with its negative log probability (Shannon, 1948), leading to the following ratio:

$$\mathbb{C}(q, \rho, f) = \frac{n_\rho^f \log q(f)}{\log q(\rho)} \tag{3}$$

where n_ρ^f counts the number of times f appears in ρ.

It is easy to verify that this definition satisfies each of our desired properties.

D1: If f is not utilized by ρ, then $n_\rho^f = 0$. Thus, $\mathbb{C}(q, \rho, f) = \frac{0 \cdot \log q(f)}{\log q(\rho)} = 0$.

D2: If the entire program is cached, then $f = \rho$ and $n_\rho^f = 1$. Thus, $\mathbb{C}(q, \rho, f) = \frac{1 \cdot \log q(\rho)}{\log q(\rho)} = 1$.

D3: Consider the case where ρ contains one instance of f. Splitting ρ into the components constituting f and those without (denoted $\rho_{\setminus f}$), we have that $q(\rho) = q(f)q(\rho_{\setminus f})$ and

$$\mathbb{C}(q, \rho, f) = \frac{\log q(f)}{\log q(f) + \log q(\rho_{\setminus f})} \tag{4}$$

As $q(\rho_{\setminus f})$ decreases, the *magnitude* of the denominator increases and hence $\mathbb{C}(q, \rho, f)$ decreases.

D4: From Eq. (4): as $q(f)$ decreases, $\log q(\rho_{\setminus f})$ contributes less, and hence $\mathbb{C}(q, \rho, f)$ increases.

The measure of the benefit in caching can be used as a tractable estimate of the benefit in chunking and hence to define our probabilistic model of $p(c \mid f, \rho, x; \boldsymbol{\phi}_\mathcal{D})$. Additionally, recall that $p(c \mid f, \rho, x; \boldsymbol{\phi}_\mathcal{D})$ should express the uncertainty in whether or not chunking f will have a positive effect on the recognition model's ability to generate ρ for *solving* x. If ρ does not solve x, then any benefit gained from chunking f to generate ρ does not matter:

$$p(c = 1 \mid f, \rho, x; \boldsymbol{\phi}_\mathcal{D}) = \mathbb{1}[\rho \Rightarrow x]\mathbb{C}(q_{\boldsymbol{\phi}_\mathcal{D}}, \rho, f) \tag{5}$$

where $\mathbb{1}[\rho \Rightarrow x]$ equals 1 if ρ solves x and 0 otherwise. As c is a binary random variable and $0 \leq \mathbb{C}(q_{\boldsymbol{\phi}_\mathcal{D}}, \rho, f) \leq 1$, the model expresses a valid probability distribution.

The expression for $p(c \mid f; \boldsymbol{\phi}_\mathcal{D})$ in Eq. (2) involves two impossible summations over the infinite sets of tasks and programs. We use a different approach to deal with each.

Marginal over tasks Expressing the joint $p(\rho, x \mid f; \boldsymbol{\phi}_\mathcal{D})$ as $p(\rho \mid f, x; \boldsymbol{\phi}_\mathcal{D})p(x \mid f; \boldsymbol{\phi}_\mathcal{D})$ and then substituting $p(x \mid f; \boldsymbol{\phi}_\mathcal{D})$ with its expression from Bayes' rule (which is proportional to $p(f \mid x; \boldsymbol{\phi}_\mathcal{D})p(x; \boldsymbol{\phi}_\mathcal{D}) = q_{\boldsymbol{\phi}_\mathcal{D}}(f \mid x)p(x)$) allows us to incorporate the recognition model's probability of generating f and, when substituted into Eq. (2), express the marginal over tasks as an expectation. The model $p(c \mid f; \boldsymbol{\phi}_\mathcal{D})$ thus becomes proportional to:

$$\mathbb{E}_{p(x)}[q_{\boldsymbol{\phi}_\mathcal{D}}(f \mid x) \sum_\rho p(c \mid f, \rho, x; \boldsymbol{\phi}_\mathcal{D})p(\rho \mid f, x; \boldsymbol{\phi}_\mathcal{D})] \tag{6}$$

We can now form Monte Carlo estimates of the expectation using the observed tasks $\mathcal{X}$. The normaliser (equal to $\mathbb{E}_{p(x)}[q_{\boldsymbol{\phi}_\mathcal{D}}(f \mid x)]$) can be estimated in the same manner.

Marginal over programs We follow the approach taken by Ellis et al. (2021) and marginalise over the finite beam of programs $\mathcal{B}_x$ maintained for each task instead. This creates a lower bound particle-based approximation. Given that only the programs that solve x contribute probability mass to the summation, excluding any programs that do not solve x has no effect on the approximation. The same is true for programs that do not utilize f. While all $\rho \in \mathcal{B}_x$ solve x, they do not necessarily contain f. We can improve the approximation by attempting to refactor each $\rho \in \mathcal{B}_x$ to a behaviourally equivalent program that instead contains f. To deal with $p(\rho \mid f, x; \boldsymbol{\phi}_\mathcal{D})$ in Eq. (6), we assume independence from f, replacing it with $q_{\boldsymbol{\phi}_\mathcal{D}}(\rho \mid x)$.

Final model We can understand $p(c \mid f; \boldsymbol{\phi}_\mathcal{D})$, our model expressing the uncertainty of whether chunking a function will benefit the recognition model's inference capabilities, as an interaction between three key sub-expressions:

$$\mathbb{E}_{p(x)}[\underbrace{q_{\boldsymbol{\phi}_\mathcal{D}}(f \mid x)}_{\textbf{(1)}} \sum_{\rho \in \mathcal{B}_x} \underbrace{\mathbb{1}[\rho \Rightarrow x] \frac{n_\rho^f \log q_{\boldsymbol{\phi}_\mathcal{D}}(f \mid x)}{\log q_{\boldsymbol{\phi}_\mathcal{D}}(\rho \mid x)}}_{\textbf{(2)}} \underbrace{q_{\boldsymbol{\phi}_\mathcal{D}}(\rho \mid x)}_{\textbf{(3)}}] \tag{7}$$

The functions with a high probability of being worthwhile to chunk are those that **(1)** first and foremost the recognition model *wants* to generate as part of programs solving some task. When it does, **(2)** chunking the function greatly reduces the uncertainty (or we could say "difficulty") in **(3)** generating the recognition model's preferred programs to solve that task. We refer to this probabilistic chunking variant of dream decompiling as DREAMDECOMPILER-PC.

5. Case Analysis: Chunking With a Uniform q

In this section we examine the use of $p(c \mid f; \boldsymbol{\phi}_\mathcal{D})$ as a chunking criterion when the recognition model distribution, parameterised by $\boldsymbol{\phi}_\mathcal{D}$, is uniform. Here, the probability of generating a function f is $1/|\mathcal{D}|^{size(f)}$. The beneficial caching measure of Eq. (3) simplifies to $n_\rho^f \, size(f) / \, size(\rho)$, representing the intuitive notion that the benefit gained in generating ρ when f is cached is simply proportionate to the number of components no longer needing to be generated. Substituting this simplification and

other uniform probabilities into Eq. (7) (with normalizer), and rearranging, we obtain:

$$p(c \mid f; \phi_{\mathcal{D}}) \propto size(f) \sum_{\rho \in \bigcup_{x \in \mathcal{X}} \mathcal{B}_x} n_\rho^f \cdot w^{(\rho)} \tag{8}$$

where $w^{(\rho)} = 1/\, size(\rho) |\mathcal{D}|^{size(\rho)}$. The benefit of chunking a function for a uniform inference distribution is proportionate to a product of the function's size and a weighted count of where the function can be used. This product matches the high-level compression objectives used by other library learning approaches (Bowers et al., 2023; Ellis et al., 2021; Cao et al., 2023; Lázaro-Gredilla et al., 2019), balancing the two properties of a highly compressive function: one that is general enough to be used frequently, yet specific enough to capture lots of functionality when used. In our case this balance arises organically when there is no knowledge to extract from the recognition model (it is uniform).

The weight associated with each function occurrence is inversely proportional to the size of the encompassing program: all else being equal, preference is given to functions within smaller programs over those in larger ones. This property is desirable for compression in this context: larger programs offer more functions (flexibility) for later compression of it and future examples.

6. Evaluation

The aim of this paper is to introduce the alternative approach for library learning and demonstrate that the knowledge learnt by the recognition model for guiding program search space can be leveraged to directly influence how to restructure the search space, via chunking, for more effective inference. We evaluate the two dream decompiler variants outlined in Section 4 to DREAMCODER across 6 program synthesis domains. Each domain was used as part of the evaluation of Ellis et al. (2021), where DREAMCODER was found to solve at least as many tasks as the best alternative tested for that domain and did so (mostly) in the least amount of time. Below, we provide a summary of each domain. For example tasks, see Fig. 4, and for further details, refer to Ellis et al. (2018; 2021).

1. *List processing*: synthesising programs to manipulate lists. The system is trained on 109 observed tasks and tested on 78 tasks. Each task contains 15 input/output examples. The initial library contains common functional programming operations and elements.
2. *Text editing*: synthesising programs to edit strings of text. The system is trained on 128 randomly generated text editing tasks (each with 4 input/output examples) and tested on 108 tasks from the SyGuS competition (Alur et al., 2017). The initial library contains most programming operations from list processing, along with a component to represent all unknown string constants.
3. *Block towers*: synthesising programs to build block towers. The system is trained on 54 observed tasks and tested on 53 held-out tasks. Tasks require constructing a target tower using an induced program that controls a simulated 'hand' to move and drop blocks.
4. *Symbolic regression*: synthesising polynomial and rational functions. The system is trained and tested with 100 functions each. All polynomials have a maximum degree of 4. Tasks include 50 input/output pairs produced by the underlying function. The initial library consists of the $+$, $\star$, and $\div$ operators, along with a component for unknown real numbers.
5. *LOGO graphics*: synthesising programs to draw images. The system is trained and tested on 80 tasks, where each is specified by a single image. The initial library contains primitives for constructing LOGO Turtle (Thornburg, 1983) graphic programs.
6. *Regexes*: synthesising probabilistic generative models. The system is trained and tested on 128 tasks each, originally sourced from Hewitt et al. (2020). Each task includes 5 strings, assumed to be generated by an unknown distribution that the system aims to infer as a regex probabilistic program.

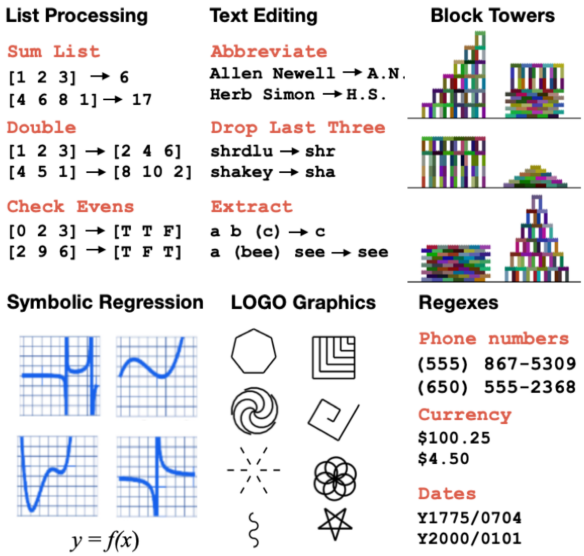

Figure 4: Example tasks from each domain tested as part of the evaluation. Figures taken from (Ellis et al., 2021).

We use the same implementation[3], architecture and settings for the main DREAMCODER model presented in (Ellis et al., 2021), unless explicitly stated. The two dream decompiling variants are evaluated within the same DREAMCODER system, with the *only difference* being the code called to update the library. This allows us to isolate the effect of chunking choice: we leave broader system changes, which may en-

[3] https://github.com/ellisk42/ec

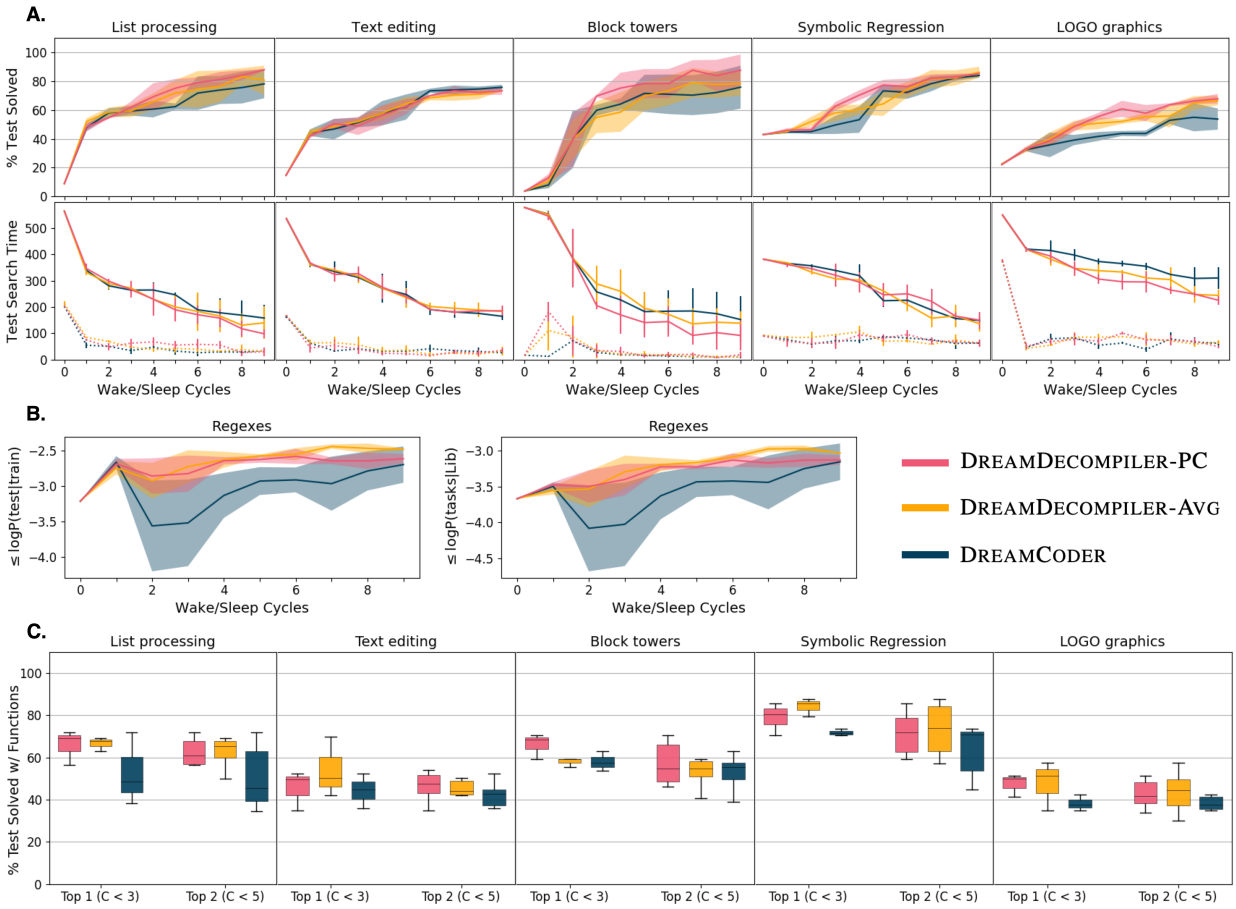

Figure 5: **(A)** Performance in deterministic program domains using the learnt library and recognition model of each wake-sleep cycle. Row 1 displays test set accuracy. Row 2 shows the average search time for test set solutions: solid lines represent the time averaged across all tasks, while dotted lines display the time averaged over solved tasks only. **(B)** Performance with probabilistic programs. The left graph shows the marginal likelihood of strings in each task, given the learnt library. The right graph shows the posterior predictive probability of held-out strings, given the programs inferred from the task's observed strings. **(C)** Percentage of test tasks solved using the top 1 or top 2 functions chunked during the first 3 or 5 cycles (C) respectively. All results are based on three random seeds with ±1 standard deviation.

hance performance in conjunction with dream decompiling, for future work (see Section 9).

The systems are evaluated on each domain for 10 of the wake-sleep cycles described in Section 2. In each cycle the system observes a random batch of training tasks during the wake-phase. The size of the batch and the time provided to search for solutions varies across domains, with smaller batch sizes and time limits used in most domains compared to Ellis et al. (2021) (see Appendix B). Harder domains place greater emphasis on chunking choice and require less computational resources. Evaluation time varies across domain, with most taking roughly half a day using 1 NVIDIA A40 and 20 CPUs (40 for LOGO graphics). For regexes and LOGO graphics, evaluations extend to over a day.

Each cycle the system is additionally tested on the domain's held-out tasks. Testing time is consistent across all domains: the system is provided 10 minutes per task to search for a solution using its current library and recognition model. Fig. 5A (Row 1) shows the percentage of test tasks solved in each cycle by all systems. Except for text editing, where performance remains comparable across all systems, utilizing the recognition model for chunking (dream decompiling) enables faster domain proficiency and enhanced generalization through the learnt library. This distinction is most evident during the intermediate cycles of learning, following similar performance in the initial iterations and before proceeding to converge again in the later iterations. Notably, this occurs despite all systems having solved a similar number of training tasks throughout (Fig. 6, Appendix C). At the respective peak differences (excluding text editing), DDC-PC outperforms DREAMCODER by 13.25% on average test

task performance in list processing and by over 17% in the remaining domains. This is achieved despite having only solved 3.7% more training tasks at the time in list processing (cycle 5) and 10.1%, 3.1%, and 2.1% more training tasks at the time in block towers (cycle 7), symbolic regression (cycle 4), and LOGO graphics (cycle 5), respectively.

The larger generalization gaps suggest that more useful components, complementing the knowledge that is learnt by the recognition model to guide search, are being extracted from the same experience. The diminishing difference indicates that dream decompiling is particularly advantageous in scenarios where fewer domain solutions (data) are available to inform chunking decisions. This advantage arises by leveraging the compiled knowledge acquired by the recognition model—knowledge obtained not only from real data but also from fantasy data. A Welch's t-test can be performed for each cycle of learning to test the null hypothesis that DDC-PC and DREAMCODER have equal average performance. We examine general performance across all domains by combining model samples from all domains and ensuring appropriate shifting to account for inter-domain differences in difficulty. This is achieved by subtracting the domain's average end performance with both models. The results show statistical significance in cycle 3 ($t : 2.57,\ p : 0.016$), cycle 4 ($t : 2.1,\ p : 0.045$), and the last few cycles, supporting claims of improved bootstrapping early on with better chunking decisions being made from the same experience.

To further test this claim, we examine the usefulness of functions chunked by each system in the earlier wake-sleep cycles. We quantify the usefulness of a function f as the percentage of test tasks eventually solved with a program containing f by the end of training. Fig. 5C (left grouping for each domain) shows the highest percentage of test tasks solved with a single function chunked by each system during the first 3 wake-sleep cycles. The right grouping shows the average percentage of test tasks solved with the top 2 functions chunked during the first 5 wake-sleep cycles. Across all domains, functions chunked early by the dream decompiling approaches are eventually utilized to solve more test tasks. This includes the text editing and symbolic regression domains, where each system solves an equal number of test tasks. Here, dream decompiling is still advantageous in making more beneficial single chunking decisions when example solutions are limited.

Fig. 5A (Row 2) shows average search time on held-out tasks, which decreases as learning progresses. By the end of learning, each system induces their program solutions (dotted lines) in near-identical average times across all domains. Notably, this holds true even in the list processing, block tower and LOGO graphic domains, where DDC-PC solves more end tasks; the combined knowledge learnt to simplify search enables access to larger relevant parts of program space *without* compromising on time—which is reflected in the lower average search times on all tasks (solid lines).

As discussed in Section 4, DDC-AVG lacks the ability to determine how many functions should be chunked; it can only rank them. Although DDC-PC addressed this limitation, determining a precise threshold for chunking a specific function still relies on unknown cost/benefit payoffs, while also accounting for the lower bound approximation. To evaluate both variants, we adopt a simple strategy of chunking the top k candidates each cycle, with k set to 2 for list processing, text editing and block towers, and 1 in the remaining domains. This provides an initial investigation, highlighting the potential of the approach to enhance learning.

Size of chunked functions Across all deterministic domains, the library learnt by DDC-PC consistently outperforms DDC-AVG, validating the use of its extra complexity. Additionally, in almost all domains the functions chunked by DDC-AVG are smaller in size than those chunked by DDC-PC (Table 1), reflecting the second issue identified with DDC-AVG in Section 4.

Table 1: Average size of chunked functions in each domain.

	LP	TE	BT	SR	LG	R
DDC-PC	5.9	7.8	8.2	4.6	5.5	3.9
DDC-AVG	5.5	8.2	6.5	3.0	4.3	3.1
DREAMCODER	5.8	6.4	8.2	4.9	5.8	3.5

7. Related Work

In this section we consider recent progress in library learning, focusing specifically on improvements proposed within, and in comparison to, the DREAMCODER system.

Library learning requires addressing two subproblems: how to (i) *generate* candidate functions for chunking, and (ii) *select* amongst the candidates for useful additions to the library. Recent advancements in candidate generation (subproblem i) are seen in BABBLE (Cao et al., 2023) and STITCH (Bowers et al., 2023): BABBLE with improved expressiveness—proposing common functionality despite syntactic differences (using domain-specific equational theories)—and STITCH with significantly improved efficiency—both time and memory use. For candidate selection (subproblem ii), both approaches adopt the compression objective from DREAMCODER—which is where we do something different, departing from compression in favour of chunking functions to complement the knowledge learnt to compose them. Our focus here was solely on selection, assuming a readily available set of candidate functions. Enhanced candidate generation from either of these approaches could thus synergize well with dream decompiling.

Wong et al. (2021) likewise introduce an alternative approach to candidate selection (subproblem ii) in library learning: an extension of DREAMCODER's compression objective that incorporates natural language annotations, modelling the hypothesis that humans often learn domain concepts with natural language descriptions.

8. Conclusion

Becoming an expert in a program synthesis domain requires learning knowledge to alleviate the search problem. DREAMCODER amortizes the cost of search by training a neural recognition model to reduce search breadth—effectively compiling useful information for composing program solutions across tasks. It reduces the depth of search by building a library capable of expressing discovered solutions in fewer components. As the library grows, a new recognition model is trained to exploit it and solve new tasks, thus bootstrapping the learning process.

Chunking program components should complement the knowledge that will be used to compose them. For this we introduced a novel approach to chunking that leverages the neural recognition model—effectively decompiling the knowledge learnt to generate program solutions across tasks and select high-level functions for the library. Consequently, the amortized knowledge learnt to reduce search breadth is now also used to reduce search depth. We show that this can lead to a stronger bootstrapping effect: better chunking decisions can be made from the same experience, leading to faster domain proficiency and improved generalization.

9. Future Work and Limitations

The experiments aimed to quantify the impact of dream decompiling by exclusively altering the selection of candidates generated by DREAMCODER. The results provide support to pursue enhanced implementations of the theory presented, including promising system-wide changes. As mentioned in Section 7, candidate generation is one. Functionality absent from existing program solutions are not generated by compression-focused approaches, yet may still score highly under dream decompiling (see Section 3). Additional approaches to candidate generation, like incorporating candidates sampled from the recognition model, are needed.

In the current system cycle (Fig. 2), the library is updated after the wake-phase, where extra information, not utilized by the recognition model for chunking, may be available. Introducing a secondary training phase, occurring after waking but before updating the library, to "compile in" this newfound information could offer further improvements.

Both variants of dream decompiling investigated can rank functions for chunking, but lack a clear means to determine an optimal number to chunk. While our experiments employed simple strategies to highlight the potential of their chunking choice, other strategies are likely more effective.

More sophisticated neural models remain an open direction for improving performance in DREAMCODER's framework. The enhanced amortization, achieved by reusing the neural model for chunking, implies that more sophisticated models could now have a cumulative impact on learning.

With dream decompiling, we are chunking functions that are useful for the *current* recognition model to generate program solutions. However, in DREAMCODER, once the library components used to specify the recognition model's distribution change, a new model is trained from scratch—with nothing learnt by the previous model passed on to the next. Exploring ways to continue learning (Kirkpatrick et al., 2017) when re-configuring its sample space could allow for improved, or at least more efficient, learning.

Acknowledgments

We are grateful to Andrej Jovanović for helpful discussions, and to anonymous reviewers for their insightful suggestions, which have greatly improved the quality of the paper.

Impact Statement

This paper presents work whose goal is to advance the field of Artificial Intelligence. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Alur, R., Fisman, D., Singh, R., and Solar-Lezama, A. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*, 2017.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Bornschein, J. and Bengio, Y. Reweighted wake-sleep. *arXiv preprint arXiv:1406.2751*, 2014.

Bowers, M., Olausson, T. X., Wong, L., Grand, G., Tenenbaum, J. B., Ellis, K., and Solar-Lezama, A. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, 2023.

Cao, D., Kunkel, R., Nandi, C., Willsey, M., Tatlock, Z., and Polikarpova, N. Babble: learning better abstractions with e-graphs and anti-unification. *Proceedings of the ACM on Programming Languages*, 7(POPL):396–424, 2023.

Chen, X., Liu, C., and Song, D. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Chollet, F. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Cox, R. T. Probability, frequency and reasonable expectation. *American journal of physics*, 14(1):1–13, 1946.

Dayan, P., Hinton, G. E., Neal, R. M., and Zemel, R. S. The helmholtz machine. *Neural computation*, 7(5):889–904, 1995.

Dechter, E., Malmaud, J., Adams, R. P., and Tenenbaum, J. B. Bootstrap learning via modular concept discovery. In *Proceedings of the International Joint Conference on Artificial Intelligence*. AAAI Press/International Joint Conferences on Artificial Intelligence, 2013.

DeGroot, M. H. *Optimal statistical decisions*. John Wiley & Sons, 2005.

Deutsch, D. *The beginning of infinity: Explanations that transform the world*. Penguin UK, 2011.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

Dumancic, S., Guns, T., and Cropper, A. Knowledge refactoring for inductive program synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 7271–7278, 2021.

Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., and Tenenbaum, J. Learning libraries of subroutines for neurally–guided bayesian program induction. *Advances in Neural Information Processing Systems*, 31, 2018.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., Cary, L., Solar-Lezama, A., and Tenenbaum, J. B. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pp. 835–850, 2021.

Ellis, K., Wong, L., Nye, M., Sable-Meyer, M., Cary, L., Anaya Pozo, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. B. Dreamcoder: growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. *Philosophical Transactions of the Royal Society A*, 381(2251):20220050, 2023.

Gershman, S. and Goodman, N. Amortized inference in probabilistic reasoning. In *Proceedings of the annual meeting of the cognitive science society*, volume 36, 2014.

Gulwani, S. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46 (1):317–330, 2011.

Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4 (1-2):1–119, 2017.

Hewitt, L., Le, T. A., and Tenenbaum, J. Learning to learn generative programs with memoised wake-sleep. In *Conference on Uncertainty in Artificial Intelligence*, pp. 1278–1287. PMLR, 2020.

Hinton, G. E., Dayan, P., Frey, B. J., and Neal, R. M. The" wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

Iyer, S., Cheung, A., and Zettlemoyer, L. Learning programmatic idioms for scalable semantic parsing. *arXiv preprint arXiv:1904.09086*, 2019.

Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., and Gulwani, S. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266): 1332–1338, December 2015. doi: 10.1126/science.aab3050. URL https://www.science.org/doi/10.1126/science.aab3050. Publisher: American Association for the Advancement of Science.

Lázaro-Gredilla, M., Lin, D., Guntupalli, J. S., and George, D. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics*, 4(26):eaav3150, 2019.

Le, T. A., Baydin, A. G., and Wood, F. Inference compilation and universal probabilistic programming. In *Artificial Intelligence and Statistics*, pp. 1338–1348. PMLR, 2017.

Le, T. A., Kosiorek, A. R., Siddharth, N., Teh, Y. W., and Wood, F. Revisiting reweighted wake-sleep for models with stochastic control flow. In *Uncertainty in Artificial Intelligence*, pp. 1039–1049. PMLR, 2020.

Liang, P., Jordan, M. I., and Klein, D. Learning programs: A hierarchical bayesian approach. In *ICML*, pp. 639–646. Citeseer, 2010.

MacKay, D. J. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

Menon, A., Tamuz, O., Gulwani, S., Lampson, B., and Kalai, A. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pp. 187–195. PMLR, 2013.

Murphy, K. P. *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023. URL http://probml.github.io/book2.

Shannon, C. E. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

Shin, E. C., Allamanis, M., Brockschmidt, M., and Polozov, A. Program synthesis and semantic parsing with learned code idioms. *Advances in Neural Information Processing Systems*, 32, 2019.

Stuhlmüller, A., Taylor, J., and Goodman, N. Learning stochastic inverses. *Advances in neural information processing systems*, 26, 2013.

Thornburg, D. D. Friends of the turrle. In *Compute!*, pp. 148, 1983.

Wong, C., Ellis, K. M., Tenenbaum, J., and Andreas, J. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, pp. 11193–11204. PMLR, 2021.

A. Handling more flexible recognition models

Chunking with either DREAMDECOMPILER variant requires calculating the probability of the recognition model generating a function as part of a program. In the main text we assumed, for simplicity, that the recognition model creates a distribution over programs by defining a distribution over library components and that the probability of the recognition model generating a program is equal to the product of independently generating each component of the program. In this case, calculating the probability of the recognition model generating a function (which is built from the same library of components) is straightforward. In this section we discuss how the probability of the recognition model generating a function can be calculated with the recognition model used by the DREAMCODER (Ellis et al., 2021) system: a bigram distribution over well-typed programs.

A.1. Well-Typed programs

All library components have an associated type specifying the entities that their computations operate on and produce. This restricts the ways in which they can be combined to form valid programs.

DREAMCODER's recognition model is constrained to be a distribution over standalone, well-typed programs of a *requested* type only. Consequently, calculating the probability of a program being generated by the recognition model involves using a type inference algorithm. After supplying a program's type, the algorithm will track which components can be used at each point in the program's construction by determining which have a return type that unifies with what is required. To get the probability of seeing a program component, the recognition model's original distribution over all library components is renormalised to one over valid components (for this part of the program) only.

If a program contains a component whose return type does not unify with the required type, or if a component of a required type is expected but missing, then the program's probability is undefined. Therefore, to use DREAMCODER's type inference algorithm to calculate the probability of the recognition model generating a function requires that it is both a well-typed and a complete program. This is not the case for all the candidate functions in $\mathcal{F}$ which are sub-expressions extracted from (refactored) programs. DREAMCODER represents programs as polymorphic typed λ-calculus expressions. We can turn each candidate into a well-typed and complete program by performing η-conversion in reverse: wrapping each candidate function in an explicit function type (lambda abstraction) with as many variables as there are missing parts. By using a modified version of DREAMCODER's type inference algorithm that ignores the newly added variables, the probability of the recognition model generating the candidate function as part of a program can then be calculated.

A.2. Bigram model

In DREAMCODER, the generative model's distribution over library components depends only on the requested type discussed in Section A.1: it is a *unigram* distribution where the probability of generating a component is independent of any surrounding program context. On the other hand, the recognition model used by DREAMCODER is a *bigram* distribution over library components, conditioning on the function that the component will be passed to, and as which argument. Therefore, the probability of a (well-typed and complete) program ρ with K components being generated by the recognition model can be seen as a product of conditional probabilities $\prod_{k=1}^{K} q_{\phi_{\mathcal{D}}}(\rho_k \mid x, (\text{pa}_k, i_k, \tau_k))$, where pa_k is component k's parent, i_k the argument index and τ_k the requested type.

We are interested in calculating the probability of the recognition model generating a candidate function explicitly as *part of* a program to solve a given task. With a unigram distribution, the recognition model could be used to score a candidate function (that has been turned into a well-typed program) directly. However, doing this with a bigram distribution would result in the probability of the recognition model generating the function as the *outermost* top-level function of a solution only—i.e. when it has no parent.

To see why this is not desirable, consider a set of tasks requiring you to manipulate lists of integers. As the only programs solving these tasks are those returning lists of integers, the only top-level functions that the recognition model is learning to infer are those returning lists of integers. The probability of the recognition model generating a function that returns, say, a Boolean (such as indicating if an element of a list is positive) would be extremely low as the initial top-level function, even if its probability of being generated elsewhere is high.

To find the likelihood $q_{\phi_{\mathcal{D}}}(f \mid x)$ of the recognition model generating a function as part of a program to solve a task we must average over all the places and ways in which the function could be generated—i.e. over all possible (pa, i, τ) triplets.

With a unigram distribution, there is no difference in where a function is generated (other than the type constraints)—and thus nothing to average over. For a bigram distribution, the same is true for all function components other than the root: once the root has been generated, the probability of the rest are conditionally independent to where in a program they are being generated. Therefore, by splitting our function into its root component f_r and remaining components $f_{\setminus r}$, our desired probability $q_{\phi_{\mathcal{D}}}(f \mid x)$ equals:

$$\mathbb{E}_{p((\text{pa}_r, i_r, \tau_r)|x)}[q_{\phi_{\mathcal{D}}}(f_{\setminus r} \mid x, f_r) q_{\phi_{\mathcal{D}}}(f_r \mid x, (\text{pa}_r, i_r, \tau_r))] \tag{9}$$

Calculating the expectations requires specifying the probability distribution $p((\text{pa}_r, i_r, \tau_r) \mid x)$ over the context in which a function could be generated. One option is to place a uniform distribution over each option that could conceivably arise: every argument of every component in the library currently being used to generate programs (along with their respective types), has equal probability. Though simple, this option has two downsides. First, one aspect motivating the use of the recognition model for chunking was to remove the human defined priors used by DREAMCODER. This option we would be doing the exact same. The second disadvantage has to do with the distribution itself: a library that contains only a few components that expect a certain type implies that all functions returning that type would be disfavoured. However, even if we expect the library to reflect some information about the tasks it is used to solve, a few library components requesting a certain type should not imply that the likelihood of generating functions returning that type should be low: those few components could be used in multiple ways.

One might be tempted, then, to use the recognition model's distribution over parent components, but this would itself require a prior over (pa, i, τ) triplets, leading to an infinite regress. Instead, we can achieve a similar effect by using a distribution over (pa, i, τ) triplets proportionate to their frequency in the program solutions that the recognition model helped discover during search. Note that the distribution in Eq. (9) is conditioned on a specific task. Therefore, it may be more appropriate to only use the programs found to solve that specific task (i.e. those in $\mathcal{B}_x$). However, a small beam size or even just low (or empty) program beams early on in learning would not provide a meaningful distribution.

B. Hyperparameters

The same base DREAMCODER system and hyperparameter values are used to compare the different approaches to chunking. The final systems differ only in the method used to update the library. In our experiments, almost all hyperparameters are set to the same value as those used by the main DREAMCODER model presented in Ellis et al. (2021). Table 2 shows the hyperparameter values (relevant to all systems) that differ in at least one domain compared to those used in Ellis et al. (2021). Additionally, DREAMCODER employs a hyperparameter, denoted as λ in Ellis et al. (2021), which controls the prior distribution over libraries. In their work, λ was consistently set to 1.5 for all domains, except for symbolic regression where it was set to 1. In our experiments, we maintain uniformity by using the same DREAMCODER model, setting λ to 1.5 for all domains.

Domain	Wake timeout (m)	Batch size	CPUs
List Processing	12	10	20
Text Editing	12	10	20
Block Towers	5	15	20
Symbolic Regression	2	10	20
LOGO Graphics	12	20	40
Regexes	6	10	20

Table 2: Hyperparameters used for experiments that differ in at least one domain from those used by the main DREAMCODER model presented in (Ellis et al., 2021), where all other hyperparameters can be found.

C. Evaluation Training Performance

Fig. 6 shows the percentage of train training tasks solved by each system for the same wake-sleep cycles shown in Fig. 5 in Section 6. Recall that as part of the DREAMCODER system (which all tested library learning approaches inherit and share), only a small batch of training tasks is provided for solving within the wake timeout during each cycle (see Table 2 for specific values). This feature imposes a limit on the improvement that can be made from one cycle to the next, in contrast

to test performance where the system attempts to solve as many held-out tasks as possible. The key takeaway from these graphs is the following: although all systems solve a similar number of training examples throughout and thus have access to the same number of training solutions for learning, the dream decompiling approaches consistently demonstrate greater generalization on the held-out tasks using the library learnt from these examples, as seen in Fig. 5 in Section 6.

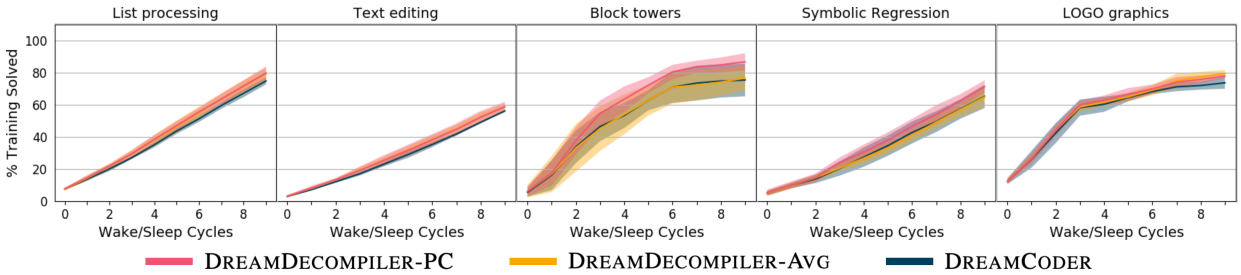

Figure 6: Performance on training tasks in deterministic program domains using the learnt library and recognition model of each wake-sleep cycle. All results are based on three random seeds with ±1 standard deviation.