A Recognition Compiler: Improving Inductive Program Synthesis with Probabilistic Chunking

Alessandro Blair Palmarini

4th Year Project Report
Artificial Intelligence
School of Informatics
University of Edinburgh

2023

Abstract

Solving program induction problems requires a search over potential solutions. Learning to solve problems efficiently means learning to simplify the search for solutions. This simplification can be achieved with knowledge to reduce either the breadth or depth of the search needed—each highly reliant on the other. DREAMCODER [10] is a program synthesis system that learns the knowledge for both in an iterative wake-sleep procedure while also solving problems. New concepts are learnt to express program solutions and a neural search policy is learnt to compose them. The neural search policy, however, has no direct effect on which concepts are formed. We extend DREAMCODER's wake-sleep algorithm with a new approach to forming concepts that instead directly leverages the knowledge learnt by the current neural search policy. We demonstrate that this new approach results in a system capable of solving a larger number of problems within short time frames upon initial exposure.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Alessandro Blair Palmarini*)

Acknowledgements

I am extremely grateful to my supervisor, Siddharth Narayanaswamy, for introducing me to, and defining, the questions that this project aimed to address. Each discussion with Sid left me wanting to learn more and continue working on similar problems in the future.

I would like to thank Antonio Vergari and Elizabeth Polgreen for being very generous with their time, listening to presentations of my work and providing useful feedback. I also thank my friend Zeno Kujawa for proofreading draft material.

Table of Contents

Chapter 1

Introduction

This chapter provides an overview of inductive program synthesis, its main challenges, and how the DREAMCODER system [10] approaches overcoming these challenges. The chapter will also provide a framing through which we can identify and understand a specific opportunity to improve the speed at which a program induction system like DREAMCODER can learn to learn. In turn, this will set up the context surrounding—and purpose for—the main ideas presented in this work.

1.1 Inductive Program Synthesis

Program synthesis refers to a large class of techniques for generating programs to satisfy certain criteria [23]. Our focus is on one area of program synthesis in which the criteria used are examples of the target program's functionality—referred to as program induction or inductive program synthesis.

The program induction problem can always 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 the inputs map to the outputs. Discovering a program that can do this—and hence generalise—can not be deduced from example behaviour: just as our scientific theories can not be deduced from observations [7]. Rather, some form of guessing is required—i.e. search.

What are we searching through? Any program must be built from some set of primitive pieces: base operations and elements. While a set of such primitives (which we now refer to as a library) are finite, the number of ways in which they can be combined, forming different programs, is infinite. Moreover, this number grows exponentially with the number of parts used to build the programs. Thus, discovering any non-trivial program using a blind search through these combinations will be unsuccessful under any reasonable resource requirement.

Instead, successfully discovering a program solution under reasonable resource constraints—and thus improving upon blind search—entails reducing the number of combinations searched before it. There are two routes to achieving this: we can (i) reduce the *breadth* searched as we move down the search tree to the target program(s) or we

can (ii) reduce the *depth* searched by pushing the target program(s) higher up the search tree.[1] Accomplishing either requires additional *knowledge*.

To reduce the breadth searched requires knowledge for avoiding search paths that do not contain the target program, while focusing on those that do—i.e. a search policy. For example, consider the program induction task specified by the input/output pair $[1,2,3] \longrightarrow [2,4,6]$. If we were to attempt to solve this task, then we may use information specific to the task—for example, in this case, that all the output elements are greater than the inputs—to avoid exploring programs that begin by decreasing the inputs, such as subtracting or dividing. Only if this was unsuccessful may we decide to explore what we initially discarded.

What knowledge is required to reduce the depth searched in finding a target program? We mentioned that all programs are built from a set of elementary units. These computational units can be combined and chained together to form new computations. What if a new computation, expressed as a series of transformations using the primitive operations, was itself turned into an elementary unit (performing the same computation) and added to the library? The unique set of computations, capable of being expressed as a program constructed from the new library, would not change: any program containing the extra unit could be replaced by one using the unit's previous constituents. However, it does mean that a program utilising this functionality can now be expressed with less components. In effect, a copy of every program containing this functionality has now been pasted higher up in the original search tree.

Therefore, to reduce the depth searched in program induction tasks requires knowledge about the functionality utilised by programs solving these tasks—allowing such functionality to be added to the library. We will refer to the act of adding a component to the library as "chunking" due to its resemblance with the cognitive mechanism in which collections of ideas are grouped into individual units [13].

There is a disadvantage to chunking. Chunking to reduce the depth required to reach computations in our search tree, simultaneously increases the breadth of reaching others: there are now more elementary units to consider. 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 (and determining what functionality to chunk for improving program induction search) is a major point of focus of this project.

The success behind many practical program induction synthesis systems (such as *FlashFill* [14]) is the careful selection of its library operations—that is, the domain-specific language (DSL) used. Human knowledge has been used to define a computational search space in which programs capable of solving specific tasks (those it is intended to solve) can be found tractably. However, this highly specialised search space implies that these systems are hopeless (or completely unable) to solve any task requiring computations of a different nature.

A general purpose inductive program synthesis system—one that is not limited in its

[1] To be clear: the tree referred to here is the tree of all program combinations, not a single program's syntax tree.

capacity to solve program induction tasks in novel domains—can not rely on fixed human knowledge. The system must be capable of learning: learning how to structure its search space (via the concepts contained in the library) and learning how to guide the search for program solutions.

DREAMCODER is an inductive program synthesis system that (beginning with its predecessor EC^2 [11]) incorporates the learning of both in an iterative *wake-sleep* algorithm—drawing on prior work (as stated in [10]) on both library learning [6, 15, 19] and learning how to guide search by training a neural network to output a probabilistic search policy [3, 8]. This enables DREAMCODER to *learn* to solve program induction tasks in many domains.

1.2 Wake-Sleep Learning

The DREAMCODER algorithm is inspired by the wake-sleep algorithm [16].

The original wake-sleep algorithm is a method for capturing the generative process that produced some observed variables (data). The observed variables are assumed to depend on unseen latent variables. The algorithm, therefore, seeks to learn a model—referred to as a generative model—that defines a joint probability distribution over these latent and observed variables.

We would like the generative model's distribution to match the true data generating distribution. Doing so can be done by minimising the Kullback-Leibler (KL) divergence between the empirical data distribution and the generative model's distribution of the data. This is equivalent to maximising the generative model's log probability of the data averaged over the empirical data distribution. However, calculating the generative model's probability of the data would require marginalising out the latent variables which, in this context, is assumed to be intractable. Therefore, the probability is approximated. In the original wake-sleep algorithm the ELBO lower bound [21] is used, but other alternatives exist such as the IWAE bound used in the reweighted wake-sleep (RWS) algorithm [4, 18]. However, approximating the generative model's marginal likelihood requires the need to approximate, and sample, from its posterior distribution—that is, the generative model's distribution over latent variables conditioned on the observed variables. This implies the need for another model—referred to as a recognition model—to act as this approximate posterior. Note that the intractability of the generative model's marginal likelihood is related to the intractability of the posterior—and hence why it must be approximated as well [17]. This is because the generative model's joint distribution over observed and latent variables is tractable and, for any random variables x and z, the posterior is equal to the joint over the marginal: $P(z \mid x) = P(x,z)/P(x)$.

In short, the wake-sleep algorithm jointly learns the generative model by additionally learning a recognition model, iterating between two phases to do so. In the wake-phase, the recognition model is held fixed and the generative model is updated accordingly to maximise its (approximate) marginal likelihood of the observed data. In the sleep-phase, the recognition model is updated to minimise the KL divergence between the true posterior under the current generative model (which is held fixed) and its

approximate posterior distribution. This amounts to maximising the recognition model's log likelihood on samples from the current generative model.

1.3 DreamCoder

As a wake-sleep algorithm, DREAMCODER can be understood as probabilistic inference problem. There is assumed to be a generative process producing unseen programs under some probability distribution. These latent variable programs express computations that give rise to the observed program induction tasks that DREAMCODER receives.

If we knew the generative distribution over programs, then we could use it to reduce our search problem for tasks in this domain. Because searching through programs in decreasing order of their probability of being generated would provide DREAMCODER the best a priori search strategy for solving both the observed and future tasks.

Unfortunately, we do not know what the generative distribution is. So we can attempt to learn an approximation instead. Approximating a distribution requires a model for that approximation. DREAMCODER models the distribution over programs using a set of library components coupled with a probability distribution over these components. The probabilities specify how likely each library component will appear in a program.

Therefore, we can find the model that is most likely to have generated our observed tasks, by looking at the functionality shared across the programs that produced those tasks. But, of course, we do not know what these programs are. We can attempt to guess what they are by finding (through search) programs that solve the observed tasks. However, as discussed in Section 1.1, to do so tractably would require extra knowledge on how to either guide or structure the search space. The generative model does both. However, using the generative model for search is problematic given that the purpose of the search here was finding an estimation for it. Moreover, if we wanted to search for programs in decreasing order of their probability of being generated, then, rather than use the generative model's prior distribution over its library components, we can keep the library components, but look at the posterior distribution of a program being generated for the specific task we are trying to solve. But, again, we do not know what this posterior distribution is so we look to approximate it with a recognition model.

These three unknowns (the generative model's library, the programs solving our observed tasks, and the recognition model's posterior distribution) motivates DREAMCODER's three step algorithm where a generative and inference model are learnt jointly in an iterative fashion (as in the original wake-sleep algorithm) while simultaneously searching for program solutions. We will briefly summarise each of these phases below.

1.3.1 Dream Sleep: Training the Recognition Model

We start with DREAMCODER's first sleep phase: where a recognition model is trained to infer programs likely to solve a given task. DREAMCODER's recognition model is a neural network providing a distribution over the generative model's library components. Training the model requires (program, task) pairs—where the program solves the task—to learn from.

DREAMCODER has two sources of (program, task) pairs that it can train on. First, as would happen in the original wake-sleep algorithm, the current generative model can be used to sample programs. After sampling a program, it can be run on example inputs to create a made-up task. This provides an unlimited stream of data to train the recognition model with. However, if we have discovered programs solving our observed tasks, then it would be wasteful not to use them. Therefore, in addition to the made-up tasks, DREAMCODER is trained to infer programs for solving any real tasks that it has observed. The recognition model is updated to maximise the same objective in both cases—the difference being that one is averaged over samples from the current generative model while the other is averaged over the empirical data distribution. Note this is similar to what is done in the extra wake-phase as part of the reweighted wake-sleep (RWS) algorithm [4, 18].

However, unlike the original and reweighted wake-sleep algorithms, DREAMCODER's recognition model is not trained to approximate the full posterior probability distribution of hidden variables conditioned on what is observed. Instead, the recognition model is updated to approximate a maximum a posteriori (MAP) estimate using the current generative model's prior distribution over programs. That is, there could be many programs (discovered to solve real tasks or sampled for made-up tasks) that all solve the same task. DREAMCODER is trained to infer only one of these programs—specifically, the most likely under the generative model—rather than them all.

1.3.2 Wake: Searching for Program Solutions

It is in DREAMCODER's wake phase where the inductive program synthesis takes places. The recognition model trained in the previous phase is used to search for as many program solutions as it can for (batches of) the observed tasks. Programs are enumerated for a provided time limit in decreasing order of their probability of solving the task according to the recognition model.[2]

1.3.3 Abstraction Sleep: Adding Functions to Library

After the wake-phase, newly discovered program solutions can be used to create a better estimate of the generative process behind the observed tasks. To do so, DREAMCODER takes a Bayesian approach: their goal is to find the most likely generative model given the observed tasks—namely, MAP inference over generative models.

The model maximising the posterior likelihood of generating the observed tasks is equivalent to that maximising the joint distribution. Calculating the joint distribution requires two components: a prior over generative models and the likelihood of a model generating the observed tasks. As the generative model provides a distribution

[2] If one is familiar with the original or reweighted wake-sleep algorithms then understanding what takes place in each of DREAMCODER's three phases, based on their naming, may be confusing. Unlike the original wake-phase, the generative model is not updated here—this, instead, takes place in the next sleep-phase. Similarly, the recognition model is not updated as in the extra RWS wake-phase mentioned in Section 1.3.1. The reason this is treated as a separate phase in their wake-sleep algorithm is because simply discovering programs that solve the observed tasks increases the lower bound approximation of the current generative model's marginal likelihood of the observed tasks–which we talk about next.

over programs, an *approximation* of the latter can be calculated by introducing and marginalising out the programs discovered during waking.[3] It is an approximation because calculating the true marginal would require an impossible marginalisation over all possible programs.

For the former, recall that the generative model is specified as both a library and a probability distribution specifying how likely each element in the library is to appear in a program. Therefore, a prior over generative models requires a prior over libraries and a prior over component distributions given a library. DREAMCODER uses a prior over libraries favouring those with functions that can be expressed in fewer components—i.e. their program syntax tree is small. For the component distributions, DREAMCODER uses a symmetric Dirichlet prior [20].

In a fully Bayesian treatment, discovering the MAP library would require integrating over the libraries' component distributions. This is intractable. Therefore, DREAMCODER uses the Akaike Information Criterion (AIC) [1] to replace the integration. Note that the AIC does not approximate integrals: it is simply a model selection criterion that will favour smaller libraries whose maximum likelihood estimate (MLE) can still account for the observations. However, with this, DREAMCODER can now score a given library.

The abstraction sleep, then, consists of searching over library space and updating the generative model to use the highest scoring library and its corresponding MLE component distribution. The search performed is a local one. First, a set of candidate functions will be produced from a refactoring algorithm that identifies common sub-components across the programs synthesised during waking [10]. Then, DREAMCODER will consider every library that results from chunking each candidate function individually into the current library, proceed with the highest scoring library, and repeat until no further improvement is possible.[4]

1.3.4 Interactions Between DreamCoder's Three Main Stages

Notice that the two sleep phases correspond to learning the two types of relevant knowledge for simplifying our search problem discussed in Section 1.1. By discovering programs that solve the observed tasks, we learn what functionality is common across programs producing our tasks and thus, by adding these functions to our library, we structure a search space where the depth required to reach programs that are likely to solve our tasks is reduced. This search space is then fed into the construction of the recognition model where it learns how to guide search through the *provided* search space in a manner adapted to a given task. Moreover, the improved library (generative model) improves this learning by offering better (program, task) pairs—that is, solutions to tasks that more closely resemble the observed tasks—to learn from. The improved search policy can then be fed to our search to solve more programs—and so on.

[3] Note that doing so also requires calculating the likelihood of a task given a program. For deterministic programs, DREAMCODER sets this to 1 if the program solves the task and 0 otherwise.

[4] DREAMCODER is initially provided a base library that is assumed to be capable—in principle—of expressing the programs performing all the computations needed to solve the observed tasks.

The three phases of DREAMCODER's algorithm form a positive feedback loop—each bootstrapping from the output of the previous phase, as illustrated in Figure 1.1.

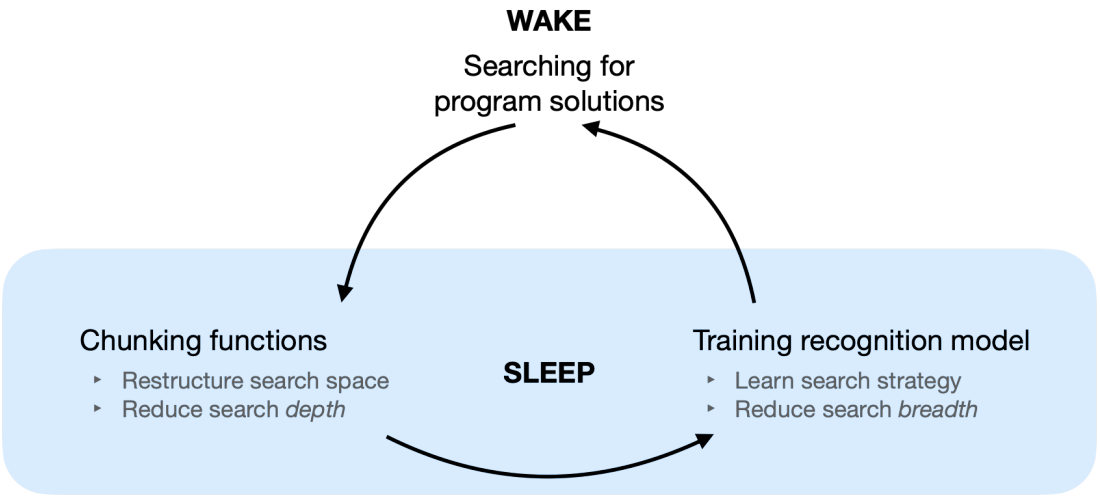


Figure 1.1: The interaction and chain of influence between each phase of DREAMCODER's wake-sleep algorithm.

1.4 A Recognition Compiler

In DREAMCODER the functions chunked are those that allow the generative model to best account for the programs synthesised during waking while conforming to the prior—and approximation—assumptions. The recognition model has only an indirect effect, then, on which functions are added to the library—based on its help in discovering the program solutions used. However, in Section 1.1, we discussed the interplay between the two pieces of knowledge (what to chunk and how to guide search) used to simplify the program induction problem. This interplay is not unidirectional.

This raises the question: can we leverage the knowledge learnt by the recognition model to influence what gets chunked—and thus, shape the search space that it will subsequently learn to guide—such that the recognition model's ability to discover program solutions is further improved?

Contrasted to DREAMCODER, the knowledge learnt from the recognition model would now directly influence, and feed into, the process chunking functions, as illustrated in Figure 1.2. This creates an inner loop between the two sleep phases. Ellis and colleagues showed that with DREAMCODER "jointly learning the library and neural search policy leads to solving more problems, and solving them more quickly" [10]. Can this new inner loop generate extra positive feedback, providing a stronger bootstrapping effect, and thus result in a system capable of learning to solve *even more* problems, and solving them *even more* quickly?

In this project we will introduce two methods that utilise the recognition model to influence chunking. We demonstrate that this approach results in a system capable of reaching a certain level of expertise in fewer wake-sleep cycles compared to DREAMCODER's approach to chunking. Moreover, in doing so, the resulting system eliminates

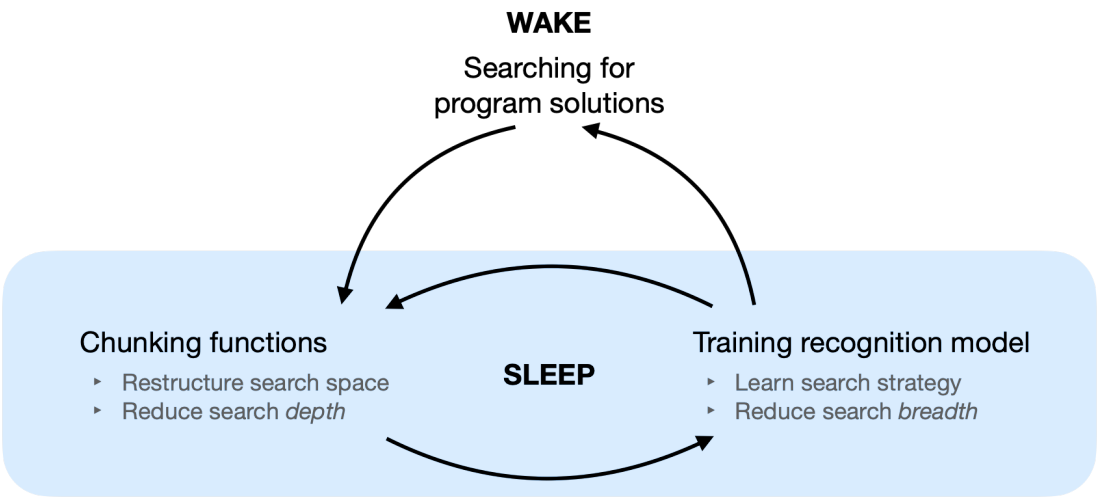


Figure 1.2: New chain of influence introduced into DREAMCODER's wake-sleep algorithm (Figure 1.1) by leveraging the knowledge learnt by the recognition model for chunking.

the need for human defined priors and a reliance on the incorrect assumption that the AIC will approximate the intractable integration, as discussed in Section 1.3.3 (and currently required in DREAMCODER's approach to chunking). Instead, the knowledge used for chunking by our methods is based on the knowledge already available in the recognition model—and hence *learnt*.

It is worth noting that the recognition model uses the same set of parameters when inferring programs likely to solve each task. This is referred to as "amortized inference" as there is shared knowledge being reused to make multiple different inferences across related tasks [12]. By additionally using the recognition model to determine what functionality to chunk—and hence made easier to use when performing new inferences—there is an even stronger amortization effect taking place.

1.5 Contributions

The main contributions of this project are the following:

- We introduce a simple approach for the recognition model to decide what is chunked by looking at which functions are most likely to be generated across tasks.
- We identify problems associated with this simple approach, and further define desirable criteria that any approach to chunking (based on the recognition model) should satisfy.
- We introduce a new probabilistic model to express the uncertainty on whether or not chunking a function will have a beneficial effect on the recognition model's ability to infer program solutions. We show how we can interpret the model and demonstrate that it satisfies all of our desired properties.
- We provide an implementation realising the ideas above within the DREAM-

CODER framework.

- We evaluate the effect of chunking choice on system performance across three program synthesis domains: list processing, text editing and symbolic regression. We demonstrate the our approach to chunking improves DREAMCODER's performance.

Chapter 2

Overview of the idea

The previous chapter explained why both chunking and having the recognition model *influence* what gets chunked could be beneficial for learning to solve program induction tasks as part of DREAMCODER's iterative wake-sleep algorithm. This chapter will present two approaches for doing so. First we introduce notation and clarify the exact computational problem.

2.1 Problem Situation

We are concerned solely with the process of determining which functions to chunk—that is, transformed into primitive operations that are added to the library and available to use directly. Note that there is an infinite number of functions that *could* be chunked. In our problem, however, we receive a finite set of candidate functions $\mathcal{F}$ available for chunking. The output, then, is a subset of $\mathcal{F}$, containing the functions which should be added to the current library.

Additional inputs include the set of observed tasks X, each program ρ found during a previous wake-phase to solve some task $x \in X$, and the current recognition model, denoted Q.

Recall that the recognition model specifies a distribution over programs likely to solve a given task. This conditional probability is written $Q(\rho \mid x)$—shorthand for $P(\rho \mid x, Q)$. Additionally, we write $Q(f \mid x)$ to denote the likelihood of the recognition model generating a candidate function $f \in \mathcal{F}$. While both programs and functions are combinations of the same library components (and indeed there may be complete programs discovered during waking that are also considered as candidate functions for chunking), it is important to note that these distributions are not identical. $Q(\rho \mid x)$ gives the probability of the program ρ being a *complete* solution to task x. On the other hand, $Q(f \mid x)$ can be understood as the probability that the recognition model generates the function f as *part* of a program solution for task x. The difference arises because the recognition model is providing a distribution over well-typed programs of a requested type only. The details are explained in Chapter 3 as they are not important (nor helpful) for understanding the approaches to chunking presented in this chapter.

2.2 Simple Approach

One approach to have the recognition model influence what functions are chunked is to simply consider what functions it wants use most often—irrespective of the particular task that it is trying to solve. That is, for any function f we can calculate the probability $P(f \mid Q)$ of f being generated by the current recognition model. This is equal to the probability of the recognition model generating f, averaged over tasks:

$$P(f \mid Q) = \sum_{x \in X} P(f \mid x, Q) P(x \mid Q) \tag{2.1}$$

$$= \sum_{x \in X} Q(f \mid x) P(x) \tag{2.2}$$

$$= \mathbb{E}_{x \sim X}[Q(f \mid x)] \tag{2.3}$$

Thus, given any set of candidate functions for chunking, we can score and rank each according to the above likelihood of being generated by the recognition model. There are two problems, however, with this simple approach. First, while a ranking can determine if chunking one function is more worthwhile than another, it provides no insight into how many functions, if any, should be chunked at all.

The second is that 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 f' of function f. The recognition model is less likely to generate f than f' because it would need to generate all components in f', plus more. Our simple approach would thus prefer to chunk the subfunction f'. However, if the recognition model is only ever intending to generate f' as part of the function f then chunking f is likely to be better: future use would require generating a single node instead of multiple and f' has no use elsewhere.

2.3 Probabilistic Model for Chunking

The simple approach provides two problems that any approach to chunking functions (that is influenced by the recognition model) would need to account for — namely, a criterion that i) determines which functions should actually be chunked and ii) considers the impact of chunking a function, rather than relying solely on generation preference. Moreover, it is unknown whether chunking a function will improve, reduce or have no effect on the recognition model's performance of helping discover program solutions. Intuitively, chunking functions will increase the available options being considered during program search. And, while some may reduce what needs discovering thereafter, others will just waste time. Therefore, to decide which functions should be chunked we must address these problems while also handling the uncertainty of not knowing what the net effect of chunking some function will be.

2.3.1 Deciding What to Chunk

A formal way to handle uncertain knowledge is with probability distributions [5]. To do this, we can first consider the Boolean random variable c that, with respect to some

function f, has the value 1 if the net effect of chunking f is positive and 0 otherwise. Then, systematically expressing uncertainty in chunking function f amounts to defining a Bernoulli distribution over c that is conditioned on the function and the recognition model: $P(c \mid f, Q)$.

With a Bernoulli distribution $P(c \mid f, Q)$ we solve the first problem of deciding whether or not a function should be chunked. For, assume that the benefit gained in a positive chunk was equal to the cost incurred in a negative chunk. Then, chunking a function f only if $P(c \mid f, Q) \geq 0.5$ would lead to the greatest payoff in expectation.[1]

More generally, given a set of candidate functions $\mathcal{F}$ for chunking, we can determine which combination of those should be chunked, and which should not, by using the individual chunking distributions $P(c \mid f, Q)$ for each $f \in \mathcal{F}$. However, in doing so we would be making the assumption that the effect of chunking some function is independent of whether or not any other functions are getting chunked. That is, for any functions $f, g \in \mathcal{F}$ we would be assuming that $P(c \mid f, Q, d_g) = P(c \mid f, Q)$ where d_g indicates the decision to have chunked function g.

In reality, it is evident that this assumption does not always hold. For example, consider again the strict subfunction case mentioned in Section 2.2. Here, there may be a positive effect from chunking either the sub or outer function individually. However, knowing that one is getting chunked could remove any benefit in chunking the other.[2] The same is true in the reverse direction. There may be functions that have no benefit in being chunked individually that would be useful if it was known that they could be used in combination with other functions being chunked.[3]

Despite this, here we only consider chunking functions in isolation, independent of any decision to simultaneously chunk other functions. This is a limitation of the approach. Future research could focus on finding ways to relax this independence assumption.

2.3.2 Capturing the Usefulness of Chunking

We now address the second problem. We want to ensure our criterion for chunking considers the overall effect on performance. As we saw, this cannot be determined based solely on which functions the recognition model wants to use (generate) most often. On the other hand, if we had access to the true distribution $P(c \mid f, Q)$ introduced previously then, by definition, our problem would be solved. Therefore, the extent to which the second problem is solved depends on how well we approximate this distribution.

The rest of this chapter will focus on presenting an approximation to this distribution that captures the probability that chunking a function will have a beneficial effect. We begin by expanding on how our distribution can be represented to further define the problem. Then, to support the approximation, we will outline desired properties of the

[1] This is true for any symmetric notion of payoff. In general, this need not be the case. Considering different notions of payoff, and how to assign them, would be an interesting area for future research.

[2] Note that this is a different problem to that being alluded to when using the example in Section 2.2.

[3] Chunking one function will always help in generating their combination. However, if both are unlikely to be generated explicitly, then the overall effect of chunking just one function may still not have an overall positive effect even if chunking both would.

distribution (and those that the true distribution is likely to have) that act as further evaluation criteria. Finally, we will introduce a probabilistic model and explain how it meets these criteria.

2.3.3 Introducing tasks

We want to approximate the beneficial chunking distribution $P(c \mid f, Q)$. This would be difficult to do in its current form. For starters, we'd need to balance the effect that chunking a function would have on the recognition model's performance across all possible tasks. Instead of handling this within the approximation, we can easily refine the problem by expanding on the distribution's representation. Specifically, we can introduce tasks as a latent variable x and marginalise them out using the sum rule:

$$P(c \mid f, Q) = \sum_{x \in X} P(c, x \mid f, Q) \tag{2.4}$$

$$= \sum_{x \in X} P(c \mid f, Q, x) P(x \mid f, Q). \tag{2.5}$$

We have, therefore, reduced the problem of needing to approximate the entire beneficial chunking distribution $P(c \mid f, Q)$, to that of approximating the distribution $P(c \mid f, Q, x)$: the probability that chunking some function will have a positive effect in helping the recognition model solve a *single* task. However, we must also address what the distribution over tasks $P(x \mid f, Q)$ is and how it can be calculated.

How should one interpret $P(x \mid f, Q)$? As a reminder, the recognition model Q is simply an approximate distribution (over which programs are the most likely to solve a given task). It is a specific model being used as a means of calculating this distribution and does not have any real influence on what tasks are observed. So, in essence, we are looking to calculate $P(x \mid f)$—with the recognition model being used to approximate its corresponding true distribution as needed.

The reason for pointing this out is that any function f can also be viewed as a stand-alone program. And, in the DREAMCODER paper [10], the authors define $P(x \mid \rho)$ as the likelihood of observing a task $x \in X$ given a program ρ: for deterministic programs this is set to 1 if and only if the program solves the task. It may appear, therefore, that we should be using the same definition for $P(x \mid f, Q)$. But here we are specifically considering f as a function used *within* other programs—not as a stand-alone program necessarily intended to solve a task in isolation. This is a crucial difference. Depending on how a task is defined, it could be the case that a stand-alone program will give rise to a single task and thus a distribution with all probability mass on that task would make sense. On the other hand, the likelihood of observing a task when conditioning on a function f used *within* the program generating the task is not going to have such a simple distribution: in most cases, the probability mass will be spread across similar tasks requiring the same functionality offered by f.

Clearly, viewing functions as programs, and using DREAMCODER's definition for the likelihood of tasks conditioned on programs, to calculate $P(x \mid f, Q)$ would be wrong.

Instead, we can use its expression from Bayes' rule:

$$P(x \mid f, Q) = \frac{P(f \mid x, Q)P(x \mid Q)}{\sum_{x' \in X} P(f \mid x', Q)P(x' \mid Q)}.$$

This allows us to make use of the approximate distribution offered by the recognition model for calculating the likelihood of functions. Additionally, we can remove Q as a conditional dependency in the task prior as it no longer depends on a program (or function) which the recognition model's approximate distribution would be used for. This leaves us with the expression:

$$P(x \mid f, Q) = \frac{Q(f \mid x)P(x)}{\sum_{x' \in X} Q(f \mid x')P(x')}. \tag{2.6}$$

Substituting in this expression to the beneficial chunking distribution (and pulling the denominator outside the sum), we obtain:

$$P(c \mid f, Q) = \frac{1}{\sum_{x' \in X} Q(f \mid x')P(x')} \sum_{x \in X} P(c \mid f, Q, x)Q(f \mid x)P(x), \tag{2.7}$$

which we can be expressed more clearly with expectations:

$$P(c \mid f, Q) = \frac{1}{\mathbb{E}_{x \sim X}[Q(f \mid x)]} \mathbb{E}_{x \sim X}[Q(f \mid x)P(c \mid f, Q, x)]. \tag{2.8}$$

There seems, at first sight, something very strange about this new expression: when calculating the probability that chunking a function f will have an overall positive effect, we are now dividing by the expected value of the recognition model generating f. Recall that this was the simple approach considered in Section 2.2. While we had identified errors in how the simple approach ranked functions for chunking, reversing this ranking is certainly no better. In fact, this seems to be doing the complete opposite of what we set out to do—namely, chunking functions in accordance to what the recognition model would find useful. However, the simple approach—that is, the expected value of the recognition model generating f—is simply the partition function for our beneficial chunking distribution: it is a normaliser that ensures $P(c \mid f, Q)$ sums to one.

Ignoring the normaliser for a moment, we can see that the probability of chunking function f having an overall positive effect on the recognition model's ability to find program solutions is proportional to an expectation under tasks:

$$P(c \mid f, Q) \propto \mathbb{E}_{x \sim X}[Q(f \mid x)P(c \mid f, Q, x)]. \tag{2.9}$$

The expectation is the product of two probabilities. First, the likelihood of the recognition model generating f for some task; and second, the probability that chunking f will have a positive effect in helping the recognition model solve that task. Thus, the functions with the highest $P(c \mid f, Q)$ are those that the recognition both wants to generate as part of task-specific solutions and, *when it does*, chunking the function would be beneficial for helping solve those tasks.

Understanding what $P(c \mid f, Q)$ is proportional to helps us make sense of the counter-intuitive partition function. The normaliser can be seen as selecting (or narrowing down) which tasks' beneficial chunking probabilities are considered: if the recognition model has no intentions of using a function on some task (low $Q(f \mid x)$), then a low $P(c \mid f, Q, x)$ on that task does not mean that the overall beneficial chunking probability $P(c \mid f, Q)$ will also be low. This is ideal, because, for a diverse enough set of tasks, we would not expect that a single function would be useful for solving *all* of those tasks.

2.3.4 Introducing Programs

In the previous section, we reduced the problem of approximating the complete beneficial chunking distribution $P(c \mid f, Q)$ to the simpler problem of approximating a task-specific beneficial chunking distribution $P(c \mid f, Q, x)$. However, even for a specific task, the effect of chunking can still depend on various factors, such as the contents and structure of programs that can solve the task—and not just those programs that utilise our function. For instance, consider the situation where a function f is to be chunked for task x. It is conceivable that even if a program using f can solve x, it may be needlessly complex compared to a simpler program solving x that does not use f. In such a case, while chunking f may lead to gains in discovering the complex program, it will also simultaneously impede the discovery of the simpler program. As a result, the net effect of chunking f for task x could be negative.

In the next section, we will examine different program scenarios in more detail to better understand when and why chunking may or may not have a positive effect. For the moment, suffice it to say that reducing the simpler problem of approximating $P(c \mid f, Q, x)$—in which we'd need to need to balance effects of chunking across multiple programs—to that of approximating $P(c \mid f, Q, x, \rho)$ for a single program ρ would be simpler still. This is easily achieved by, again, introducing programs as a latent variable and marginalising them out using the sum rule:

$$P(c \mid f, Q, x) = \sum_{\rho} P(c, \rho \mid f, Q, x) \tag{2.10}$$

$$= \sum_{\rho} P(c \mid f, Q, x, \rho) P(\rho \mid f, Q, x). \tag{2.11}$$

Thus, the probability that chunking some function will have a positive effect in helping the recognition model solve a single task is equal to a weighted average—according to the distribution $P(\rho \mid f, Q, x)$ over programs—of the probability that chunking the function will have a positive effect in helping the recognition model solve that task with a *particular* program. We will return to what the distribution over programs $P(\rho \mid f, Q, x)$ is in Section 2.3.8.2.

2.3.5 Defining the Probabilistic Model for Chunking

So far we have only used standard laws of probability to help define our beneficial chunking distribution $P(c \mid f, Q)$ that we want to approximate. This allowed us to express it in terms of the more specific distribution $P(c \mid f, Q, x, \rho)$. At this point, further refinements are neither likely to be helpful for approximating the target distribution nor,

to all appearances, possible. Therefore, we are left wanting to capture the Bernoulli distribution over the random variable c representing the uncertainty that chunking function f would be useful for helping the recognition model Q generate program ρ for solving task x.

This is certainly an easier problem than what we began with. However, even for a single program and a single task, there is a difficult trade-off that remains to be addressed. Consider what happens when a function is chunked. The function is not just added to the library as a single-node operation. Instead, the function is added to the library and *then* the recognition model is updated, defining a new distribution over programs in terms of the newly available library.

On its own, knowing that function f is part of program ρ solving some task x, does not guarantee that there will be a positive effect (in terms of helping the recognition model generate ρ) from chunking f. If a function was chunked and the recognition model's updated distribution remained unchanged (thus allocating 0 probability mass to the functions newly added library node) then there would be no effect in chunking f for that particular program and task at all. Similarly, consider what would happen if the function was chunked and all of the recognition model's probability mass was shifted onto its node, such that $Q(f \mid x) = 1$. In this case, there would be a huge (maximal) improvement in generating f, but the overall effect of chunking f to help generate ρ would be detrimental as, unless $\rho \equiv f$, the rest of ρ would never be generated. Thus, the effect of chunking f—for synthesising a particular program ρ on a given task—depends on a balance between the improvement gained in generating f minus the costs incurred in generating the rest of ρ as a result.

Note that the value of c here could be measured exactly: we could chunk f, retrain the recognition model, and then check for any performance gains in generating ρ on task x. But this is equivalent to trying each option as a means of deciding which option to take. If this was computationally feasible then there would be no need for finding a beneficial chunking distribution to help make our decision in the first place. How, then, can we efficiently express our uncertainty in whether or not chunking a function will have a beneficial effect?

We can not calculate the effect of *chunking* f for generating ρ on task x without knowing how the recognition model is updated. However, there is a related effect that we can both capture precisely and calculate efficiently: the effect that *caching* f has on helping the current recognition model generate ρ on task x. That is, how helpful it is for the recognition model to generate ρ given that it doesn't need to generate f. In terms of the trade-off mentioned above, this is the extreme (optimistic) view that chunking f will lead to maximal improvement in generating f while, simultaneously, not incurring any cost for generating the rest of ρ.[4]

Moreover, it is possible to quantify how useful caching a function is for the recognition model on a scale ranging from 0 (not useful at all) to 1 (as useful as can be). Therefore,

[4]Note that the word "caching" is used here to simply summarise the interpretation: when a value is "cached" it implies that the normal work involved in its calculation can be avoided. We are not talking about the effect of literally caching the function into a hash table.

this lends itself as an excellent estimate, and proxy, for our uncertainty in whether or not chunking a function will have a beneficial effect.

What properties would this caching measure have? We discuss these next and then introduce a quantity that satisfies these criteria.

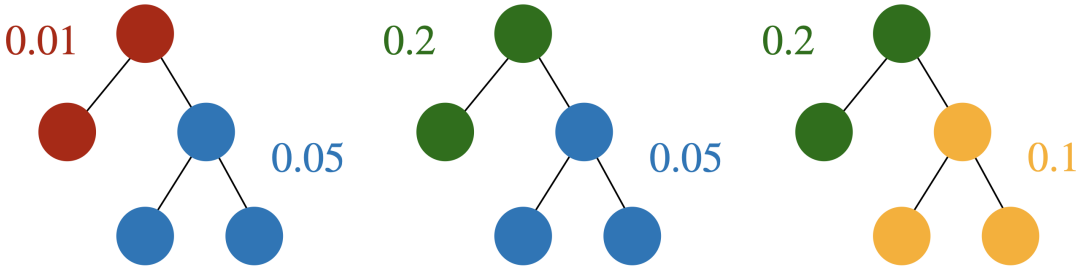
2.3.6 Desired Properties

We want a measure, ranging from 0 to 1, of how useful caching a function f is for the recognition model Q generating the program ρ to solve task x. This, in turn, will serve as our approximation for $P(c \mid f, Q, x, \rho)$. For this reason, in what follows, we use $P(c \mid f, Q, x, \rho)$ to refer to both the caching measure and our beneficial chunking distribution.

Considering the purpose of the measure, it seems desirable that it would have the following properties:

1. The first, trivial property, has nothing to do with our function. Our goal is to measure how useful caching f is for helping Q generate ρ to *solve* x. Therefore, if ρ does not solve x, then $P(c \mid f, Q, x, \rho)$ should be 0.
2. Similarly, if f is not used in ρ, then caching f provides no benefit and $P(c \mid f, Q, x, \rho)$ should be 0.
3. On the other hand, the largest benefit would be gained from caching the entire program no matter what the recognition model's distribution is. Thus, if the function f equals ρ, then $P(c \mid f, Q, x, \rho)$ should be 1.
4. The fourth property concerns how useful it is to cache a function across differing programs. Consider a function f part of two programs ρ_1 and ρ_2, both of which solve our task x. We are dealing with the same task and thus the probability $Q(f \mid x)$ of the recognition model generating f within each program is constant. However, if the probability of the recognition model generating the rest of ρ_1 is much smaller then that of ρ_2 (as with the example programs of Figure 2.1a and 2.1b), then caching the same function f is less useful for Q needing to generate ρ_1 compared to ρ_2. (As an extreme example, if the probability of generating the rest of our program became 0, then caching f can not help in generating the full program at all.)

   Therefore, the less likely it is for the recognition model to generate the outer program, the less useful caching a fixed function would be—and hence, the smaller $P(c \mid f, Q, x, \rho)$ should be. (For our example programs, we would want $P(c \mid f, Q, x, \rho)$ to be higher with the function-program pair found in Figure 2.1b over that found in 2.1a.)
5. The fifth property concerns how useful it is to cache different functions used within the same outer program. Consider the same outer program ρ_o capable of using one of two different functions f_1 or f_2 to solve our task x. In this scenario the probability of Q generating the outer program is what remains constant. Assume that the recognition model is less likely to generate f_1 versus f_2 (as



(a) Low outer likelihood (b) Reference likelihoods (c) High function likelihood

Figure 2.1: Three example programs. In each example the top left nodes are considered as the outer program and the bottom right subtree as the function being cached. Both the outer program and function are labeled with their respective probabilities of being generated according to the recognition model.

with the example programs of Figure 2.1b and 2.1c). Would it be more useful to cache f_1 for generating its full corresponding program $\rho_o \circ f_1$ or to cache f_2 for generating $\rho_o \circ f_2$? We know that after caching, both programs are equally likely to be generated. However, had the functions not been cached, then $\rho_o \circ f_1$ would have been less likely to have been produced than $\rho_o \circ f_2$. Hence, for their respective goals, caching f_1 provides more benefit than caching f_2.

This property can be stated more concisely as follows: The less likely it is for the recognition model to generate a function within the same outer program, the more useful caching that function would be—and hence, the higher $P(c \mid f, Q, x, \rho)$ should be. (For our example programs, we would want $P(c \mid f, Q, x, \rho)$ to be higher with the function-program pair found in Figure 2.1b over that found in 2.1c.)

Note that a quantity satisfying the last two properties would resolve the strict subfunction problem we encountered when chunking functions according to the recognition model's raw generation preference (simple approach) in Section 2.2. For, given a fixed program and task, when considering to chunk any function over one of its strict subfunctions, we are consuming more nodes from the outer program into the function. Consequently, the likelihood of generating the outer program will increase and thus, as per property 4, the usefulness measure would increase. Additionally, the likelihood of generating our function would decrease and thus, as per Property 5, the usefulness measure would, again, increase.

In fact, the last two properties are two sides of the same coin. Both suggest that the usefulness in caching a function f—for the purpose of helping the recognition model generate a target program ρ—is related to the *relative proportion of uncertainty removed* in generating ρ by caching f. It turns out that, with the right interpretation, this points us to the quantity needed for satisfying the above criteria exactly.

2.3.7 Measure of Caching Usefulness

We will now define a quantity that satisfies the desired properties listed above, measuring how useful caching a function f is for the recognition model Q generating a program ρ to solve task x. As mentioned in Section 2.3.5, this quantity serves as our probabilistic model for the distribution $P(c \mid f, Q, x, \rho)$ representing the belief that chunking f would have an overall positive effect—which, in turn, is sought after because discovering this value exactly is computationally infeasible. The model can be expressed as follows:

$$P(c \mid f, Q, x, \rho) = \mathbb{I}[\rho(x)] \frac{\log Q(f \mid x)^n}{\log Q(\rho \mid x)}, \tag{2.12}$$

where $\mathbb{I}$ is an indicator returning one or zero, $\rho(x)$ denotes whether or not the program ρ solves the task x, and n is the number of times the function f is used within the program ρ.

Let us return to the conclusion of Section 2.3.6: that the usefulness in caching a function f depends on the relative proportion of uncertainty removed when caching f to generate a target program ρ. This serves as an excellent interpretation of the model presented above. In information-theoretic terms, "uncertainty" is often used synonymously with the "self-information" or "surprise" inherent in a random outcome [25]—such as a function or program being generated by the recognition model. An outcome that is guaranteed to occur has no uncertainty surrounding whether it will or will not be generated. On the other hand, the less likely a random outcome is to occur, the more uncertainty there is associated with it. Capturing the "uncertainty" of a random outcome formally can be done with the negative log probability under which it occurs.

Therefore, the uncertainty with a program ρ being generated by the recognition model would be equal to $-\log Q(\rho \mid x)$. We can then ask: how much of this uncertainty is associated with the production of a function f? Answering this question can be done using our expression for $P(c \mid f, Q, x, \rho)$ in Equation 2.12.

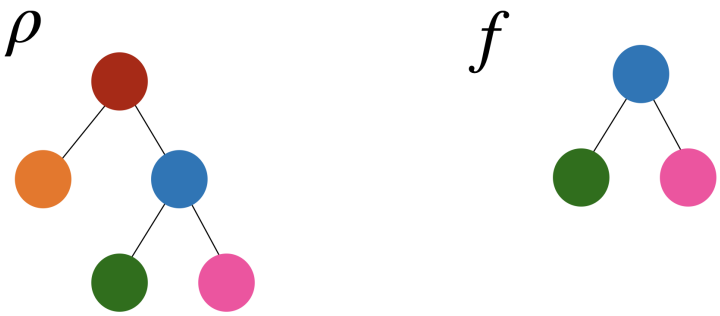


Figure 2.2: An example program and function used within the program.

To aid understanding of the model, consider the example program and function illustrated by their program trees in Figure 2.2. Each colour represents a different primitive operation. We can see that the program on the left contains the function on the right as a sub expression. If, for simplicity, the probability of the recognition model generating a program was equal to the product of its parts, then, assuming ρ solves the given task, the usefulness in caching f for helping the recognition model generate ρ would be

calculated using the following ratio:

$$P(c \mid f, Q, x, \rho) = \frac{\log Q(\bullet \mid x) + \log Q(\bullet \mid x) + \log Q(\bullet \mid x)}{\log Q(\bullet \mid x) + \log Q(\bullet \mid x) + \log Q(\bullet \mid x) + \log Q(\bullet \mid x) + \log Q(\bullet \mid x)}$$

We will now verify that this model satisfies each of our desired properties (as a measure of caching usefulness) in addition to being a valid probability distribution for its role as a probabilistic model.

2.3.7.1 Caching is useful only if ρ solves x

The first property required that $P(c \mid f, Q, x, \rho) = 0$ if ρ does not solve task x. This is handled by the indicator function $\mathbb{I}[\rho(x)]$, where we have used $\rho(x)$ to denote whether or not ρ solves task x. If ρ solves x then we are left with the rightmost fraction; otherwise, if it doesn't, then $P(c \mid f, Q, x, \rho) = 0$ as required.

2.3.7.2 Caching is useful only if ρ uses f

The second property required that $P(c \mid f, Q, x, \rho) = 0$ if f is not used in ρ. This is clearly satisfied because if f is not part of ρ, then $n = 0$ and thus (assuming that ρ solves x) we have that

$$P(c \mid f, Q, x, \rho) = 1 \cdot \frac{\log Q(f \mid x)^0}{\log Q(\rho \mid x)} = \frac{\log 1}{\log Q(\rho \mid x)} = 0.$$

2.3.7.3 Maximum benefit when caching the full program

We wanted our measure to have a maximum value of 1 and to occur when the function being cached was equal to our program—that is, $f = \rho$. If this is the case then $n = 1$ and (assuming again that ρ solves x) we have that

$$P(c \mid f, Q, x, \rho) = 1 \cdot \frac{\log Q(f \mid x)^1}{\log Q(\rho \mid x)} = \frac{\log Q(\rho \mid x)}{\log Q(\rho \mid x)} = 1.$$

2.3.7.4 Less useful when caching in less likely outer programs

Our fourth property required that the less likely the recognition model is to generate the outer program, the less useful caching a fixed function should be. If the program ρ contains one instance of our function f, then ρ can be broken up into two parts: the outer program, which we will denote ρ_o, and the function f. The probability that the recognition model generates the entire program ρ can be expressed as a product of generating the outer program and the function:

$$Q(\rho \mid x) = Q(\rho_o \mid x) Q(f \mid x). \tag{2.13}$$

If we substitute this expression into our model, where we know $n = 1$ and are assuming that ρ solves x, we get the following:

$$P(c \mid f, Q, x, \rho) = \frac{\log Q(f \mid x)}{\log Q(\rho_o \mid x) + \log Q(f \mid x)}. \tag{2.14}$$

Expressed like this, it is clear how our definition of $P(c \mid f, Q, x, \rho)$ satisfies the fourth desired property. We are considering the same function f part of different outer programs and thus the numerator is the same across all cases. However, the less likely the recognition model is to generate the outer program ρ_o, the larger the absolute value of the denominator becomes and hence the smaller $P(c \mid f, Q, x, \rho)$ becomes—as desired.

2.3.7.5 More useful when caching less likely functions

The fifth property required that, all things being equal, the less likely the recognition model is to generate a function, the more useful caching that function should be for generating the entire associated program. We can confirm that our model satisfies this property by using our expression of $P(c \mid f, Q, x, \rho)$ stated in Equation 2.14 again. In this case, the outer program remains constant. Thus, as the probability of the recognition model generating our function decreases, the absolute value of the log probability increases and the less that the log probability of the recognition model generating the outer program contributes to the denominator. Consequently, the measure of how useful it is to cache f will increase as desired.

2.3.7.6 Valid probability distribution

We now confirm that $P(c \mid f, Q, x, \rho)$ is a valid probability distribution. As c is a binary random variable, this amounts to ensuring that any value output by the model is between zero and one.

It is easy to see that $P(c \mid f, Q, x, \rho)$ will always be greater that or equal to zero. The reason being that $Q(\cdot \mid x)$ is a probability in $[0, 1]$ and thus $\log Q(\cdot \mid x)$ is always nonpositive. Moreover, as $n \geq 0$ and $\mathbb{I}[\rho(x)] \geq 0$, the sign of the numerator can not be flipped. Therefore, we are dividing a nonpositive value by another nonpositive value and hence $P(c \mid f, Q, x, \rho) \geq 0$.

What about $P(c \mid f, Q, x, \rho)$ always being less than or equal to 1? Well, if f is a function contained *within* ρ, then the probability $Q(f \mid x)$ (or any number of function uses within the program) will always be greater than or equal to $Q(\rho \mid x)$. Consequently, the magnitude of its log probability will be smaller and hence $P(c \mid f, Q, x, \rho) \leq 1$. The only situation where this would not be the case is if $Q(f \mid x) < Q(\rho \mid x)$. However, this can only possibly arise if f is a function not used in ρ. And, as we saw in Section 2.3.7.2, in this case $n = 0$ and $P(c \mid f, Q, x, \rho) = 0$. Therefore, $0 \leq P(c \mid f, Q, x, \rho) \leq 1$ as desired.

2.3.8 Completing the Probabilistic Model

In Section 2.3.7 we defined a probabilistic model for $P(c \mid f, Q, x, \rho)$. This expressed the uncertainty in the belief that chunking a function f will have a beneficial effect in aiding the recognition model solve a given task using a given program. We can now take one step back and substitute our model into the task-specific beneficial chunking

distribution stated in Equation 2.11:

$$P(c \mid f, Q, x) = \sum_{\rho} \mathbb{I}[\rho(x)] \frac{\log Q(f \mid x)^n}{\log Q(\rho \mid x)} P(\rho \mid f, Q, x) \tag{2.15}$$

Before taking the final step back in specifying our probabilistic model, there are two more issues needing to be addressed.

2.3.8.1 Marginal over programs

Our expression for $P(c \mid f, Q, x)$ currently involves marginalising over all programs ρ. Unfortunately, the set of all programs is infinite and thus summing over each element is impossible.

This is not a new problem: training of the recognition model and DREAMCODER's approach to chunking functions both involve marginalising over the set of all programs. DREAMCODER handles this by using a particle-based approximation. That is, rather than marginalise over the infinite set of programs, a finite set of programs (referred to as a beam) is used instead. A unique beam of programs $\mathcal{B}_x$ is maintained for every task $x \in X$, each containing the programs found during waking to solve that task. We take the same approach here. Doing so gives us a lower bound approximation for our task-specific beneficial chunking distribution:

$$P(c \mid f, Q, x) = \sum_{\rho \in \mathcal{B}_x} \mathbb{I}[\rho(x)] \frac{\log Q(f \mid x)^n}{\log Q(\rho \mid x)} P(\rho \mid f, Q, x). \tag{2.16}$$

Which programs should be used within a finite sized beam if we want to maximise our lower bound approximation? Due to the indicator $\mathbb{I}[\rho(x)]$ (which is there to satisfy the first desired property of $P(c \mid f, Q, x, \rho)$ given in Section 2.3.6), we know that if a program does not solve our task, then it it will not contribute any probability mass to the summation. Excluding such programs does not impact the accuracy of our lower bound. For our purposes, then, it only makes sense to maintain programs in a beam $\mathcal{B}_x$ that solve task x—as DREAMCODER already does. However, as per the second desired property of $P(c \mid f, Q, x, \rho)$, we also know that if a program does not incorporate the candidate function being evaluated for chunking, then it, too, will not contribute any probability mass to the summation.

Consider the following example. We want to find the probability $P(c \mid f, Q, x)$ that chunking the function f will be beneficial, where f is the function $f(a) = a + a$ that takes as input a number a and doubles it. Assume that the only program in our beam $\mathcal{B}_x$ is $\rho() = 1 + 1$. As ρ does not use f, the probability that chunking f could be beneficial will be zero. However, we know that in this case chunking f is likely to be beneficial: if f was part of the library, then we could generate the functionally equivalent program $\rho'() = f(1)$ using two components rather than three. Indeed, had the program that first constructs f explicitly and then calls it with the input 1 been part of the original beam instead, then the approximated $P(c \mid f, Q, x)$ would not have been zero.

Thus, when considering a finite number of programs $|\mathcal{B}_x|$ in a given beam, there always remains an opportunity to improve our lower bound approximation by searching for a

refactored program that is functionally equivalent—thus not losing its ability to solve our task—whilst also incorporating our candidate function if it did not already. Doing so can also be relatively inexpensive compared to what would have been needed to synthesise the program incorporating our function from scratch. Note that considering refactored programs part of the beam to improve our lower bound particle-based approximation is an idea taken directly from DREAMCODER.

Let $\rho_{[f]}$ denote program ρ refactored to use function f. The probability distribution representing the uncertainty in the belief that chunking a function f will have a beneficial effect in aiding the recognition model solve a given task x can then be expressed as follows:

$$P(c \mid f, Q, x) = \sum_{\rho \in \mathcal{B}_x} \mathbb{I}[\rho(x)] \frac{\log Q(f \mid x)^n}{\log Q(\rho_{[f]} \mid x)} P(\rho_{[f]} \mid f, Q, x). \tag{2.17}$$

2.3.8.2 Distribution over programs

We now address what should be used to calculate the probability distribution over programs—now denoted as $P(\rho_{[f]} \mid f, Q, x)$—that we left in Section 2.3.4. Of course, we do not have access to this distribution. It is, after all, a posterior distribution over programs conditioned on a task: the reason for having a recognition model and what it is trained to approximate. However, this particular posterior is also conditioned on a function f. There are two ways we can approach this.

The first is to assume independence. That is, we assume that the function f has already been generated as (or part of) another program and we now want to find the probability of the new program $\rho_{[f]}$—where what has previously been generated has no effect.[5] Therefore, with this approach we have that $P(\rho_{[f]} \mid f, Q, x) = Q(\rho_{[f]} \mid x)$.

Recall that in Section 2.3.3 we took $P(x \mid f, Q)$ to mean the probability of observing a task given that the function f was used as part of the generative process (program) that produced it. The second approach to calculating $P(\rho_{[f]} \mid f, Q, x)$ is to take the same view, only this time we are looking at the posterior distribution over the full generative process (program) rather than tasks. In this view, the probability of any program not utilising function f should be zero. We can attempt to calculate the distribution using Bayes' rule. Doing so would require another impossible sum over all programs. However, given that we are already working with a finite beam of programs, we can consider the posterior over these programs only:

$$P(\rho_{[f]} \mid f, Q, x) = \frac{P(f \mid \rho_{[f]}) Q(\rho_{[f]} \mid x)}{\sum_{\rho \in B_x} P(f \mid \rho_{[f]}) Q(\rho_{[f]} \mid x)}, \tag{2.18}$$

where $P(f \mid \rho_{[f]})$ is the likelihood that f was part of the generated program $\rho_{[f]}$—which equals 1 if and only if $\rho_{[f]}$ uses f—and is conditionally independent of x and Q.

With the first approach, for each program in the beam, $P(c \mid f, Q, x, \rho_{[f]})$ is weighted by the likelihood of the recognition model generating $\rho_{[f]}$ relative to all other programs. On the other hand, with the second approach, $P(c \mid f, Q, x, \rho_{[f]})$ would be weighted

[5] This is the same assumption made by choosing the family of distributions that can be represented by the recognition model considered here and in DREAMCODER.

by the likelihood of the recognition model generating $\rho_{[f]}$ relative only to the other (refactored) programs in our beam $\mathcal{B}_x$ utilising f.[6]

Preliminary experiments found the second approach to perform worse than the first. This is presumably because too much information is lost when normalising the likelihood of the recognition model generating the programs containing the candidate function for chunking. Thus, from her on out we stick with the first approach. Equation 2.17 can thereby be restated as:

$$P(c \mid f, Q, x) = \sum_{\rho \in \mathcal{B}_x} \mathbb{I}[\rho(x)] \frac{\log Q(f \mid x)^n}{\log Q(\rho_{[f]} \mid x)} Q(\rho_{[f]} \mid x). \tag{2.19}$$

2.3.9 Final interpretation

At the start of this chapter we faced the problem of finding a criterion for chunking functions that could leverage the knowledge learnt by the recognition model. We approached the problem by seeking a probability distribution $P(c \mid f, Q)$ that could allow us to systematically express the uncertainty of whether or not chunking a function would have an overall positive effect on the recognition model's ability to help aid discovery of program solutions. Having addressed the remaining issues with our task-specific distribution, we can now take the final step back, substituting our expression for $P(c \mid f, Q, x)$ in Equation 2.19 into that of $P(c \mid f, Q)$ in Equation 2.8:

$$P(c \mid f, Q) = \frac{1}{\mathbb{E}_{x \sim X}[Q(f \mid x)]} \mathbb{E}_{x \sim X}[Q(f \mid x) \sum_{\rho \in \mathcal{B}_x} \mathbb{I}[\rho(x)] \frac{\log Q(f \mid x)^n}{\log Q(\rho_{[f]} \mid x)} Q(\rho_{[f]} \mid x)]. \tag{2.20}$$

Although daunting at first, we can attempt to understand $P(c \mid f, Q)$ as an interaction between three key sub-expressions:

$$P(c \mid f, Q) \propto \mathbb{E}_{x \sim X}[\underbrace{Q(f \mid x)}_{\textbf{(1)}} \underbrace{\sum_{\rho \in \mathcal{B}_x} \mathbb{I}[\rho(x)] \frac{\log Q(f \mid x)^n}{\log Q(\rho_{[f]} \mid x)}}_{\textbf{(2)}} \underbrace{Q(\rho_{[f]} \mid x)}_{\textbf{(3)}}], \tag{2.21}$$

where we have dropped the normaliser for clarity. According to this expression, 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. The functions that are chunked are those that can successfully balance this interaction across tasks.

[6] Note that not all attempts at refactoring a program in the beam will result in a program containing our function.

Chapter 3

Further Details

The ideas discussed in previous chapter were presented at a high level for ease of understanding. Implementing the ideas in practice, however, requires addressing several additional details and challenges. To be clear, the implementation done in this work consisted in extending the openly available DREAMCODER codebase.[1] The resulting system is still largely, and mainly, the work of DREAMCODER: it differs only in the code called to update the generative model's library—that is, deciding which functions to chunk—in each wake-sleep iteration. Nonetheless, changing this one component requires that the implementation operates successfully withing the surrounding framework. The codebase is extensive, containing over 13000 lines of Python code: understanding how concepts are represented, what functionality is available, and how the different parts of the system interact presented a significant initial challenge. Furthermore, a deeper understanding of the low-level details revealed several challenges overlooked by the previous chapter's high-level presentation. This chapter reviews the most significant of these challenges and how they were overcome.

3.1 Probability of Functions

Both approaches to chunking considered in Chapter 2 require calculating the probability of the recognition model generating a *function* as part of a program solution for a given task. Thus far we have 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. If this was the case, then applying the recognition model to functions (which are built from the same library of components) would be straightforward. However, this is not the case: two further details make calculating the probability of the recognition model generating a function more involved.

[1] `https://github.com/ellisk42/ec`

3.1.1 Types

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.

While it's possible to receive ill-typed candidate functions, the main problem is that many of them—as *sub-expressions* extracted from programs—have missing parts. This is not a problem for chunking, but the type inference algorithm can not be used. One solution is to wrap the candidate function in an explicit function type (if it is not already) with as many variables as there are missing parts.[2] The resulting program would be well-typed and complete—and hence, its probability of being generated can be calculated. However, this would also contain the probability of generating each additional variable. Thus, we can deal with this by using dummy variables instead and modifying DREAMCODER's type inference algorithm to ignore them (for being generated and in contributing to any renormalisations), such that running the probability calculation would return the desired result: the probability of the recognition model generating the candidate function explicitly.

3.1.2 Bigram Model

In DREAMCODER, the generative model's distribution over library components depends only on the requested type discussed in Section 3.1.1: 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 $\prod_{k=1}^{K} Q(\rho_k \mid x, (\mathrm{pa}_k, i_k, \tau_k))$, where pa_k is component k's parent, i_k the argument index and τ_k the requested type.

[2]Note that DREAMCODER represents programs as polymorphic typed λ-calculus expressions: the procedure described is η-conversion in reverse.

Recall that we are after the probability of the recognition model generating a candidate function explicitly as *part of* a task solution. In Section 3.1.1 we discussed how we can get around the type inference algorithm to do so. However, this would only suffice for a recognition model using a unigram distribution. For, with a bigram distribution, the result would be the probability of the recognition model generating our function as the *outermost* top-level function of a solution only—i.e. when it has no parent.

This is not desirable. Consider a set of tasks requiring you to manipulate lists of integers. Because 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.

For the simple approach (Section 2.2), when we say we want the probability of the recognition model generating a function for a given task, what we are really after is the function's probability of being generated *averaged* over all the places and ways in which it could appear—specifically, 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. The same is true for all function components other than the root in a bigram distribution: 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_{root} and remaining components f_{rest}, our desired probability is

$$Q(f \mid x) = \mathbb{E}_{(\text{pa},i,\tau)}[Q(f_{\text{rest}} \mid x, f_{\text{root}})Q(f_{\text{root}} \mid x, (\text{pa}, i, \tau))] \tag{3.1}$$

$$= Q(f_{\text{rest}} \mid x, f_{\text{root}}) \sum_{(\text{pa},i,\tau)} Q(f_{\text{root}} \mid x, (\text{pa}, i, \tau))P((\text{pa}, i, \tau) \mid x). \tag{3.2}$$

This raises the question: what is the probability distribution $P((\text{pa}, i, \tau) \mid x)$ over the context in which any function could be generated? We consider two options.

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, as discussed in Section 1.3.3. With 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 mean that the chances to use functions returning that type should be low: those few components could be used (or generated) very often.

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—and hence lead

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 programs that the recognition model helped discover during waking. Note that the distribution in Equation 3.2 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 distribution suited for its purpose.

Calculating the probability of the recognition model generating a candidate function as part of a program for solving a given task can thereby by summarised as follows. First, a sweep over all programs discovered during waking counts the occurrences of (pa, i, τ) triplets.[3] Next, a function is transformed into a well-typed program using dummy variables that will be ignored by the type inference algorithm—affording the opportunity to find the probability of generating the function components only. The likelihood of all function components other than the root can be calculated separately. This is then multiplied by the likelihood of the recognition model generating the function's root component, marginalised over all (pa, i, τ) triplets—considering only those where the function's return type unifies with τ. This is not a large nor expensive marginal to perform.

3.2 Beneficial Chunking Distribution

The extra details concerning well-typed programs (Section 3.1.1) and contextual recognition models (Section 3.1.2) presents additional challenges (and notable differences) when it comes to calculating the beneficial chunking distribution laid out in Section 2.3.

First, recall that the probabilistic model used for the task and program specific chunking distribution $P(c \mid f, Q, x, \rho)$, as first defined in Equation 2.12, contains the log probability $Q(f \mid x)$ of the recognition model generating the provided function, multiplied by each use n in the program. The previous section explained how we can calculate $Q(f \mid x)$. However, additionally recall that the probabilistic model was intended to capture the relative proportion of uncertainty removed when generating the *particular* program ρ without the need to generate the function f. Thus, unlike previously (where we wanted the likelihood of the function being generated as part of any program intended to solve a task), we do not want to marginalise out the contextual information on where a function is generated. Rather, we seek the specific contextual probability of f being generated as required in the provided program ρ.[4] Additionally, as multiple function occurrences may appear in different contexts, the likelihood of each occurrence may be different. And thus, one must sum over these log probabilities rather than multiply a single value by the number of occurrences n.

[3] This only needs to be performed once as it can be reused for each candidate function.

[4] If we were dealing with monomorphic types and unigram distributions (as assumed in Chapter 2), then the likelihood of any function being generated by the recognition model would be equivalent for any context—and hence, there would be no need to differentiate between the two distributions. Perhaps it would have been more appropriate to introduce some notion of context (in which a function can be used) in Chapter 2 and then indicate when it was being marginalised out.

The next challenge is in calculating the other half of the probabilistic model's ratio: the log probability of the current recognition model generating the provided program. As discussed in Section 2.3.8.1, the programs considered here are the solutions discovered during waking, refactored to contain our candidate function for chunking. DREAMCODER's refactoring algorithm is used to do so. However, a successful refactoring will return a program constructed using the function as a primitive operation. Therefore, as the candidate function is not part of the recognition model's library, the likelihood of this program being generated is not defined.

Note that the same would be true if the primitive operation was replaced with the candidate's explicit function definition: in this case it is still an enclosed function operation (λ abstraction) that, to be validly used in a program, would need to be part of the library. The space of programs that the recognition model places a distribution over does not include those that define new function operations to use within them—even if the functions are composed of the same library components.

Despite this, the computations expressible by programs in the recognition model's sample space are the same as those in the space of programs containing explicit function definitions. For, a program containing an explicit function definition can always be transformed into one that does not: by applying the arguments to the function (i.e. eliminating the function with β-reduction).[5] Thus, by eliminating the function in this manner, we have a program whose probability can be calculated using the current recognition model. However, this is *not* what we want: to find the proportion of uncertainty removed when generating a program that doesn't need to generate the candidate function, we need to first know the probability of generating the program with the candidate function. The transformed program may perform the same computation, but it does not use the candidate function. Moreover, the transformed program may be radically different to the one that does: for example, a function using an argument multiple times will result in a program tree that repeats the argument sub-tree multiple times.

To capture the desired effect, we must stick with the refactored program that defines our candidate function while using it. But, given that this program is not part of the recognition model's sample space, we will consider a counterfactual scenario: what is the probability of the recognition model generating the refactored program given it gets to generate an inner program (function definition) for each of occurrence of the candidate function—i.e. generate the candidate functions and the remaining program separately.[6]

We know how to calculate the probability of the recognition model generating the candidate function. With a unigram distribution, we would also be able to calculate the probability of the outer program. These can then be combined for the desired result. With a bigram distribution, on the other hand, there is yet another problem: any

[5] This would be undoing the work done by DREAMCODER's refactoring algorithm which is performing inverse β-reductions [10].

[6] Note that if programs containing new function definitions were part of the recognition model's sample space in the first place, then this roundabout solution would not be needed. Thus, it may be worthwhile to explore using recognition models where this is the case in future work.

arguments of a candidate function are considered part of the outer program, but since the function is not in the library, the recognition model has no conditional distribution over library components with the candidate function as the parent. To address this, a distribution approximating the likelihood of the recognition model generating these components is needed. One solution is to use the uniform distribution, but a better option in this case would be to use the generative model's unigram distribution instead.

In practice this is implemented using another modified version of DREAMCODER's type inference algorithm. A program is traversed as normal, tracking type requests and other context details at each program component. However, when a candidate function is reached, the type request and context is used to calculate its probability of being generated as in Section 3.1.2, then skipped over.[7] Additionally, if a bigram recognition model is used, the root components of any candidate function arguments are calculated using the generative model.

3.3 Merging Beams

After determining which candidate functions to chunk and adding them to the library, there is another issue that must be dealt with before the next phase of the system's cycle can be ran. Recall that DREAMCODER maintains a finite beam of programs $\mathcal{B}_x$ for each task x. For every candidate function f, while calculating the task-specific beneficial chunking distribution $P(c \mid f, Q, x)$, the same beam of programs in $\mathcal{B}_x$ will be refactored (if possible) to contain f. Thus, if we decided to chunk M functions, then we will have $M \cdot |\mathcal{B}_x|$ programs available that solve x. Note that this may contain more than one copy of a program, arising when a program part of the original beam could not be refactored for at least 2 of the chunked functions.[8] However, even after eliminating duplicates, the total number of programs may exceed the beam's capacity. Hence, we must decide which of these programs are used to fill each task's beam for the next phase.

One option is to maintain a single beam and use DREAMCODER's refactoring algorithm to refactor the programs in it for each chunked function in order of their rank, starting with the highest-ranked function. That is, rather than refactor the original beam separately for each chunked function, we use the refactored beam of the first function as the starting point for the next, and so on. However, DREAMCODER's refactoring algorithm relies on data structures set up initially when finding the candidate functions [10]. After performing one refactoring, the algorithm would be unable to operate on the resulting program because it contains a new library component (the first function we have chunked) that it is not prepared to deal with.

To achieve the above, we would need to rerun the part of DREAMCODER's algorithm that constructs the data structures used for refactoring. However, doing so would be very costly. (Note that the inability to carry out this option may be a consequence of

[7] A cache is kept mapping type requests to the whats needed to calculate a function's likelihood efficiently so that the type inference algorithm for our candidate function doesn't need to be rerun across programs.

[8] If we want to explicitly consider the original programs that do not contain any of our chunked functions, then we could have a potential $(M+1) \cdot |\mathcal{B}_x|$ programs available for soling x.

attempting to use DREAMCODER's refactoring algorithm as an unmodified black box. We suspect that the same result could be achieved without requiring complete reruns by adapting the algorithm to our needs. However, no easy approach to doing so was found and, given it is unrelated to the main problem addressed in this project, it was not considered a top priority.) Additionally, all existing references to our chunked functions would be replaced, requiring a search for the new ones.

Instead, to set a task's beam, the following approach was taken. First, note that the original programs part of the beam are already ordered by DREAMCODER according to their likelihood of being generated by the generative model. Then, using DREAMCODER's refactoring algorithm, a refactored version of this beam is produced separately for each function being chunked, producing $M \cdot |\mathcal{B}_x|$ programs as discussed above. Next, the programs are interleaved such that the first M programs are the M different refactorings of the first program and ordered according to the respective function's chunking score used for the refactoring—starting with the highest. The next M are for the second program, and so on. Finally, while ignoring duplicates, the programs are added to the returned beam in this order until the maximum capacity is reached.

The main downside to this approach is that any program that may have been refactored by the first option to simultaneously utilise more than one of the chunked functions, won't be made part of our beam used in training the next iteration of the recognition model. However, this approach was deemed preferable as it avoids rerunning an expensive computation multiple times for a situation that may not arise often in practice. Additionally, the double refactored program always has a chance to be dreamt or discovered during the next waking phase.

After updating the library and stored programs to make use of the newly chunked functions, we can then updated the generative model by setting its distribution over the library components to their MLE of generating the updated programs. (This was done by calling code available from DreamCoder as it is something used in their approach as well.)

Chapter 4

Evaluation

The aim of this project is to identify if the knowledge learnt by the recognition model for guiding program search space can be leveraged to directly influence how to restructure the search space (through chunking program components), such that more problems can be learnt to be solved more quickly. We do so by evaluating the effect of chunking choice across the following three program induction task domains. (Each domain is taken from [11] and similarly used in [10] for evaluating DREAMCODER.)

1. **List processing**: synthesising programs to perform computations on lists. The system is trained on 109 observed tasks and tested on 78 held out tasks.[1] Each task is comprised of 15 input/output examples. Examples of these tasks are dropping the first three elements of a list with observed input/output pairs such as $[0,3,8,6,4] \rightarrow [6,4]$ or counting the number of times the first element appears in the rest of the list with observed input/output pairs such as $[1,2,1,1,3] \rightarrow 2$. The system is given an initial library containing the following functional programming operations and elements: `fold`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, `is-nil`, `mod`, `*`, `>`, `is-square`, and `is-prime`.

2. **Text editing**: synthesising programs to manipulate strings of text. The system is trained on 128 randomly generated tasks (each comprised of 4 input/output examples) and tested on the 108 tasks used in the 2017 SyGuS program synthesis competition [2]. Examples of the tasks are extracting someone's initials with observed input/output pairs such as "Nancy FreeHafer" $\rightarrow$ "N.F." or reversing the order of a name with observed input/pairs such as "Launa Withers" $\rightarrow$ "Withers Launa". The system is provided the same initial library as in list processing, although it does not contain the last five operations listed above (`mod`, `*`, `>`, `is-square`, and `is-prime`). Additionally, the library contains a primitive `string` for representing all unknown string constants—which is then replaced by the longest common string subsequence occurring in the observed examples [10].

3. **Symbolic Regression**: synthesising programs to capture polynomial and rational functions. All polynomials have a maximum degree of 4. An example task

[1] These numbers differ from [11] because we use the same tasks used in [10] where 21 tasks equivalent to the identity function and 31 test tasks solvable by the base system were removed.

is representing the function: $f(x) = -1.9x^4 + 1.9x^3 - 2.0x^2 - 0.5x + 2.4$. The system receives 50 input/output pairs produced by the underlying function as well as a visual graphic of its curve. The system's initial library consists of the addition, multiplication and division operators as well as a primitive for representing any unknown real numbers (analogous to `string` in text editing). The program synthesis problem is that of finding the main structure of the hidden function and then a gradient descent procedure is used to set all the unknown real numbers used in a program.

We begin by running the main DREAMCODER model of [10]. For the most part, all hyper parameters and model architectures used are the same as those used for the full DREAMCODER model reported in [10]. There are a couple differences. First, in our experiments only 20 CPUs are used, compared to the 64 used by [10] in the list processing and text editing domains. Second, in [10] one of their hyper parameters (that denoted as λ in [10]) for controlling the prior over libraries is set differently in the symbolic regression domain. We, instead, keep this consistent with the value used in the list processing and text editing domains. This ensures the same DREAMCODER model is used across all domains. The only difference across domains is the time spent attempting to solve tasks during wake-phases: the list processing and text editing domains have a 12 minute time limit, whereas a 2 minute time limit is used for symbolic regression—as in [10].

In each domain we run the system three times with different random seeds for ten of the wake-sleep cycles described in Section 1.3. At the start of each cycle the system is tested on the domain's held out tasks. The system is provided 10 minutes per task to search for a program solution—using its current library and recognition model. (Note that no information from testing the system is fed back into impacting the next iteration's training.) The left column of Figure 4.1 displays the percentage of these testing tasks that the system was able to solve (within the 10 minute time limit) at each wake-sleep iteration. The right column of Figure 4.1 displays the time spent (in seconds) by the system searching for a solution: the solid line shows the time averaged over all testing tasks, whereas the dotted line shows the time averaged only over the tasks that the system was able to solve. In both sets of figures the average over each run is reported with error bars indicating the ± 1 standard deviation points. Note that in the work of [10], DREAMCODER is compared to numerous baselines: in every domain DREAMCODER was found to solve at least as many tasks as the best alternative for that domain and does so, for the most part, in the least amount of time.

To test the approaches to chunking presented in this project, we run the same DREAMCODER system modified only in the method used to chunk functions. We do so for both the simple approach and the beneficial chunking distribution (labeled as RECOGNITIONCOMPILER) presented in Sections 2.2 and 2.3 respectively. All hyper parameters used are the same as those used when running the main DREAMCODER model in the respective domain. Therefore, one should refer to [10] if after any specific details about the resulting system that are not related to the selection of functions being added to the library.

Recall that chunking functions can be disadvantageous: you decrease the search depth

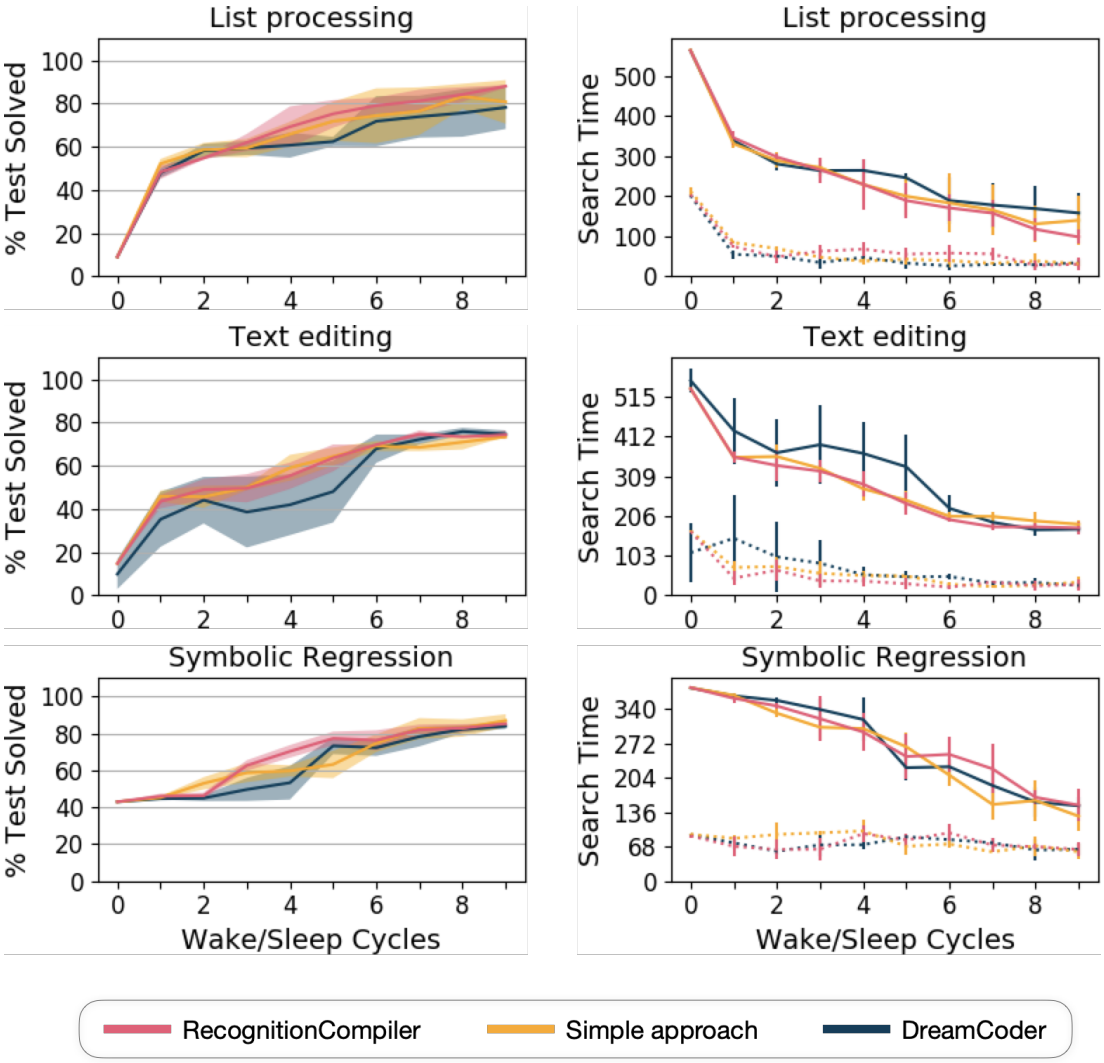


Figure 4.1: **(Left)** Test set accuracy using the learnt library and recognition model of each wake-sleep iteration. **(Right)** Time spent (seconds) searching for solutions to the test set tasks: solid lines show time averaged over all tasks and dotted lines show time averaged over solved tasks. Results over three random seeds with ± 1 standard deviation error bars. (Graphs produced with code available in `https://github.com/ellisk42/ec`.)

needed to discover programs performing particular computations while also increasing the search breadth for discovering others. Therefore, simply deciding to add, or not add, any function to the library could have an effect on performance. We want to test if the knowledge learnt by the recognition model can be used to make better decisions on what to chunk for improving skill in a domain. Hence, to eliminate additional factors that may influence performance, for the list processing and text editing domains we match the *quantity* of functions being chunked by DREAMCODER, allowing us to focus on *quality*. Specifically, when running our modified systems, we rank and chunk the top candidate functions to match the amount chunked by DREAMCODER on that iteration when run using the same random seed.

On average, DREAMCODER chunked 15.33 and 14 total functions across all iterations of

a single run in the list processing and text editing domains respectively. Thus, matching these numbers provides lots of opportunities to compare the effect in the quality of functions chunked by the different approaches. On the other hand, in the symbolic regression domain, DREAMCODER only chunked an average of 2.67 total functions: for most iterations it deemed the chunking of any candidate function to be detrimental. Thus, in this case, matching the number of functions chunked would provide very little room for comparison and for either approach to be clearly distinguished as better. For this reason, when running the simple approach and beneficial chunking distribution in the symbolic regression domain we use a simple strategy of chunking one function (the highest ranked) per iteration instead.

In all domains we see the same trend between DREAMCODER's approach to chunking and the beneficial chunking distribution. Both systems achieve similar performance towards the end of the 10 wake-sleep cycles. However, using the beneficial chunking distribution to decide what functionality is chunked results in better performance when given less wake-sleep cycles to learn. After 4 wake-sleep cycles in symbolic regression, RECOGNITIONCOMPILER was capable of solving, on average, 17% more problems than DREAMCODER. For the list processing and text editing domains, the peak difference occurs at iteration 5. Here RECOGNITIONCOMPILER solved, on average, 12.8% (list processing) and 15.6% (text editing) more tasks than DREAMCODER. [2]

Additionally, on all domains, using our probabilistic model for chunking generally outperforms the recognition model's generation preference (simple approach), validating the use of its extra components.

[2]Note that one run of DREAMCODER in the text editing domain is substantially different from the other two. If this seed is removed then performance between the two systems is much closer—with the RECOGNITIONCOMPILER solving only 3.2% more tasks then DREAMCODER.

Chapter 5

Discussion

Becoming a domain expert in program synthesis tasks requires learning knowledge to alleviate the search problem. Broadly, this can be achieved by reducing the breadth or depth of the search needed. DREAMCODER does both: by training a neural search policy (recognition model) to guide search and chunking common functionality utilised in program solutions, allowing it to express computations in fewer components. In DREAMCODER the neural search policy has no direct say on what functionality is chunked. Rather, the functions chunked are those contributing to the library that can best express current program solutions while adhering to provided priors and assumptions made by approximation choices. In this project we demonstrate that one can instead leverage the knowledge learnt by the neural search policy to decide which functions are chunked (and hence made available to use as part of its search policy), improving the speed at which new tasks can be learnt to be solved.

We began with a simple approach that looked at which functions the recognition model was most likely to generate when averaged over observed tasks. We identified several problems with this approach. We showed how these problems could be resolved with a probabilistic model expressing the uncertainty in whether or not chunking a function would have a beneficial effect on the recognition model's ability to help solve similar tasks. Experimental results corroborated these theoretical arguments. However, many directions remain open to build upon this approach.

5.1 Autonomy

When testing the recognition model based approaches to chunking on the list processing and text editing domains, the number of functions chunked was determined by the quantity chunked by DREAMCODER. This was to ensure a fair comparison between the quality of functions chosen for chunking. However, one could rightly criticise these approaches as being incomplete as they only provide half of the solution: determining *what* to chunk, but relying on DREAMCODER to decide *how many*. Indeed, this was the first issue we had with the simple approach in Section 2.2. While we resolved this with the beneficial chunking distribution in Section 2.3.1, a complete solution to chunking is still contingent on cost/benefit payoffs associated with chunking a specific function

while, additionally, taking the lower bound approximation into account. Thus, without DREAMCODER's help in suggesting how many functions to chunk, performance when using these approaches may drop.

On the other hand, an analogous argument could be made in which matching the number of functions chunked by DREAMCODER is seen as a constraint, limiting the potential of these approaches. Indeed, we saw this to be the case in the symbolic regression domain with even the simplest of methods for deciding how many functions to chunk.

The work done in this project serves as a preliminary investigation that hopefully motivates the potential in leveraging the knowledge learnt by the recognition model to influence what to chunk. There will be many ways to turn either of the approaches presented here into autonomous systems. For example, using fixed probability thresholds or fixed quantities, functions determining either based on input variables such as library size or function type, or even looking at relative scores between candidates. Some will be better than others. Exploring these options would be of interest if looking to push the capabilities of these approaches—and hence the speed at which autonomous systems can adapt to solving novel program synthesis tasks—as far as possible.

5.2 Recognition Model

There are several open directions regarding the recognition model and its training.

First is the use of more sophisticated neural recognition models. This was an open goal for extending DREAMCODER's ancestor EC^2 [11], with the intent that it could help extend their approach from writing small programs to synthesising large bodies of code. The approaches presented in this work leverage the knowledge learnt by the recognition model for reducing search breadth, to also reduce search depth. Thus, more sophisticated models may now provide a compound benefit.

Second, recall from Section 1.3.1 that DREAMCODER's recognition model is trained to infer the most likely program to solve a task (MAP inference) rather than approximate the full posterior distribution—as in the original wake-sleep algorithm. This was done so that the recognition model would learn to infer only one of many possible solutions, allowing needlessly complex programs (like those multiplying values by 1, for example) to be ignored. Now that the recognition model influences what is chunked, it would be interesting to compare the effect that the recognition model's training objective has on which functions are chosen to be chunked.

A key motivation behind using the recognition model to influence what gets chunked is that its knowledge for guiding search may have something to say about how to structure the search space that it guides, such that it can find program solutions quicker. However, in the current systems, after functions are chunked, a new recognition model is trained from scratch—with nothing learnt by the previous recognition model passed on to the next. Exploring ways to maintain, or make use of, the knowledge learnt by the recognition model when re-configuring its sample space may allow for more improved, or at least more efficient, learning.

5.3 Connections to Biological Sleep

The original wake-sleep algorithm drew inspiration from the learning that could be occurring in biological sleep (hence the name). DREAMCODER's wake-sleep algorithm is speculated to go a step further, bringing wake-sleep algorithms as a whole "closer to the structure of actual learning processes that occur during human sleep" [9]. The key reason is the use of two separate sleep phases, each of which has a correspondence to a separate stage of biological sleep. Human slow-wave sleep (deep sleep) is associated with the consolidation of new declarative information [24], analogous to what takes place when functions are chunked in DREAMCODER's first sleep stage. Human REM sleep (dream sleep) is associated with consolidating implicit skill through the use of memories and dreams [9], analogous to what takes place when training the recognition model in DREAMCODER's second sleep stage.

There is, however, an important difference between the high-level resemblance: humans will cycle between their sleep phases four to six times in a single night [22], whereas DREAMCODER will only visit each sleep phase once before revisiting the wake phase. Moreover, attempting to resolve the difference with a simple modification that has DREAMCODER reenter its abstraction sleep phase, after training the recognition model in its dream sleep phase, would not suffice. The reason being that functions chunked by DREAMCODER depend only on the programs discovered during waking. Thus, without undergoing another wake phase, it makes no sense to undergo additional sleep phases.

In the work presented here, our modified versions of DREAMCODER also only visit each sleep phase once before revisiting the wake phase. However, now the recognition model has a direct influence on what is chunked. Therefore, unlike DREAMCODER, it could make sense to cycle between the two sleep phases (as it happens in human sleep) before revisiting the wake phase. After training a new recognition model there may be new functions worthy of chunking and thus, if chunked, require updating the recognition model again to operate in the newly structured search space. Note that this corresponds to moving around the inner feedback loop explained in Section 1.4 and illustrated in Figure 1.2. Would doing so lead to a greater ability at learning to solve new program induction tasks quickly, with fewer attempts spent searching for solutions during waking phases? This question is interesting in its own right, but with the connection to biological sleep it becomes even more interesting.

Bibliography

[1] Hirotugu Akaike. A new look at the statistical model identification. *IEEE transactions on automatic control*, 19(6):716–723, 1974.

[2] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*, 2017.

[3] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[4] Jörg Bornschein and Yoshua Bengio. Reweighted wake-sleep. *arXiv preprint arXiv:1406.2751*, 2014.

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

[6] Eyal Dechter, Jonathan Malmaud, Ryan Prescott Adams, and Joshua B Tenenbaum. 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.

[7] David Deutsch. *The beginning of infinity: Explanations that transform the world*. Penguin UK, 2011.

[8] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[9] Kevin Ellis. *Algorithms for learning to induce programs*. PhD thesis, Massachusetts Institute of Technology, 2020.

[10] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 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*, pages 835–850, 2021.

[11] Kevin M Ellis, Lucas E Morales, Mathias Sablé-Meyer, Armando Solar Lezama, and Joshua B Tenenbaum. Library learning for neurally-guided bayesian program induction. 2018.

[12] Samuel Gershman and Noah Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the annual meeting of the cognitive science society*, volume 36, 2014.

[13] Fernand Gobet, Peter CR Lane, Steve Croker, Peter CH Cheng, Gary Jones, Iain Oliver, and Julian M Pine. Chunking mechanisms in human learning. *Trends in cognitive sciences*, 5(6):236–243, 2001.

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

[15] Robert John Henderson. *Cumulative learning in the lambda calculus*. PhD thesis, Imperial College London, 2013.

[16] 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, 1995.

[17] Diederik P Kingma, Max Welling, et al. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019.

[18] Tuan Anh Le, Adam R Kosiorek, N Siddharth, Yee Whye Teh, and Frank Wood. Revisiting reweighted wake-sleep for models with stochastic control flow. In *Uncertainty in Artificial Intelligence*, pages 1039–1049. PMLR, 2020.

[19] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, pages 639–646. Citeseer, 2010.

[20] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022.

[21] Kevin P. Murphy. *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023.

[22] Aakash K Patel, Vamsi Reddy, and John F Araujo. Physiology, sleep stages. In *StatPearls [Internet]*. StatPearls Publishing, 2022.

[23] Armando Solar-Lezama. Introduction to Program Synthesis (Lecture 1), 2018.

[24] Matthew P Walker. The role of slow wave sleep in memory processing. *Journal of Clinical Sleep Medicine*, 5(2 suppl):S20–S26, 2009.

[25] Wikipedia contributors. Entropy (information theory) — Wikipedia, the free encyclopedia, 2023.