# Program Synthesis with Symbolic Properties

by

Theodoros Sechopoulos

B.S. Computer Science and Engineering, Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
January 19th, 2022

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Joshua B. Tenenbaum
Professor
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

# Program Synthesis with Symbolic Properties

by

Theodoros Sechopoulos

Submitted to the Department of Electrical Engineering and Computer Science
on January 19th, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Program synthesis is the task of automatically writing computer programs given a specification for their behavior. Program synthesis is challenging due to the combinatorial nature of the search space. In the short term, improving program synthesis could make people vastly more productive, by transforming how they communicate with computers. In the long term, improving program synthesis could bring us a step closer to understanding human intelligence and to building machines with human-like intelligence. In this work we discuss how symbolic properties (which are themselves programs) can help program synthesis performance. Specifically, building on the formulation of properties in Odena and Sutton (2020) we present *PropsimFit*, a novel online synthesis algorithm that uses properties for program search and show that it outperforms naive non-property baselines in the Rule (2020) list function dataset. Finally, we discuss future ways to use properties for synthesis based on the insights gained from *PropsimFit* and its limitations.

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor

## Acknowledgments

I would like to thank Max Nye for introducing me to research and passing on (directly and indirectly) valuable advice on how to be effective in research: what questions to ask, how to run experiments, how to analyze and keep track of results, how to structure meetings, and the list goes on. I also owe a lot of gratitude to Evan (Yewen Pu). Apart from always keeping meetings fun, Evan has taught me a lot through his general but deep observations about pretty much anything, ranging anywhere from program synthesis to Chinese calligraphy. I also want to thank Josh Tenenbaum. Josh has always been kind and understanding while supporting me and providing valuable advice for whichever research direction I've been most interested in following without imposing a strict agenda. Additionally, his work and his views have deeply influenced how I think of human cognition. I would also like to thank Augustus Odena for volunteering his time and valuable advice in weekly meetings where we discussed my work on properties. I've also benefited from briefly collaborating with Cathy Wong, learning important organizational skills and good practices for running experiments. I also want to thank Matt Bowers for really fun and inspiring conversations about properties, AI and research more generally. Finally, although I haven't directly collaborated with Kevin Ellis, I've benefited tremendously from his work and Dreamcoder codebase which I've relied on for most of my research this past year. In the same spirit I would like to thank Josh Rule both for creating the dataset my experiments rely on, but also more generally for his thesis work which has influenced my thoughts on modelling human cognition.

# Contents

# List of Figures

# List of Tables

# 1 Why program synthesis

Program synthesis is the task of automatically writing computer programs given a specification for their behavior. Program synthesis technology could have many benefits. In the short term, improving program synthesis could make people vastly more productive, by transforming how people communicate with computers. In the long term, improving program synthesis would bring us a step closer to understanding human intelligence and to building machines with human-like intelligence.

Technically, there are several methods by which a user can specify the program that they want synthesized. In one common method, the desired programs are specified by examples of input-output pairs, that is, the expected outputs that are produced when executing the program on specific inputs. Alternatively, natural language descriptions of the program's intent are also often used. Program synthesis is often formulated as a search problem over possible combinations of elementary program primitives of a Domain Specific Library (DSL). Even the best synthesis systems can only synthesize relatively short programs due to combinatorial explosion. The size of the search space gets prohibitively large as the length of the allowed programs increases. In this work, we use symbolic programs to represent properties of programs and discuss how they can help address the above issue and advance program synthesis.

## 1.1 Transforming how we interact with computers

Fundamentally, for a computer to have any effect on the world it must execute code. The way we communicate to computers is by writing code. To facilitate code writing we have invented programming language of increasing level of abstraction. For example if we want to have the computer calculate what 3 times 19 is we can just write and execute the following python code "3 * 19". This is interpreted into byte-code, which is then interpreted into machine code which is a set of instructions the CPU can directly process and execute to give the answer "57" (after converting back from binary). Communicating to the computer in python is a lot easier than communicating to the computer in machine code. The most common way we communicate with computers is by interacting with Graphical User Interfaces (GUI). Clicking around in an application corresponds to specific computer code being executed. It is this code that the user indirectly "writes" and communicates to the computer. For example even if one does not know how to code, they can have the computer calculate what 3 times 19 is by opening their calculator app and clicking on the button "3", then "$*$", then "1", then "9", and finally =.

### 1.1.1 Increased software engineering productivity

High-level programming language, operating systems, applications and websites and all the components that allow users to interact with a computer are created by writing code. Since people rely on specialized applications and websites to make use of computers, how useful computers currently are depends on the quality and amount of such specialized application and websites. Program synthesis has the potential of facilitating the job of software engineers, making the process of writing code faster, which would allow for more and better specialized applications enabling users to make even better use of computers. In fact, program synthesis tools that aim to make software engineers more productive already exist, with Github Copilot (a tool based on the synthesis model Codex (Chen et al., 2021)) being perhaps the most well-known such example.

### 1.1.2 Replacing user interfaces

Program synthesis has the potential of drastically changing and in some cases even replacing costly specialized user interfaces. It takes time to learn how to use some user interfaces[1]. Additionally it is costly to build the user interface on top of the API / DSL of a specific application. It is often easier for the user to express what

[1]Which is why people put skills like Excel use on their resume (similar to how they might have programming language)

they want to do either with natural language or input-output examples. FlashFill (Gulwani, 2011) is a notable example of a real-world input-output synthesis system that facilitates the use of Excel, whereas voice assistants are examples of real-world program synthesis systems from natural language specifications. Thus developing program synthesis should broaden the usefulness of computers by facilitating the use of specialized applications. More ambitiously, advanced synthesis systems could completely replace user interfaces, expanding the scope of what computers can do for people. In summary, program synthesis has the potential to greatly improve the usefulness of computers, by facilitating the communication between user and computer.

## 1.2 Understanding human intelligence

### 1.2.1 Modelling human intelligence

Any discipline that studies the natural world needs theories that make testable predictions. In the same way, to gain insight into human intelligence we want to build computational models of intelligent human behavior. For a given intelligent human behavior, we can build a computation model that receives the same inputs as people, and compare its output behavior to that of humans. The way in which these differ can then illuminate aspects of human intelligence that our current theories of human intelligence fail to capture. With better theories we can build better computational models which can capture increasingly complex intelligent behaviors that previously seemed out of reach. The above could be useful regardless of whether the mind is fundamentally a computational system as argued by proponents of the Computational Theory of Mind (CTM) (Rescorla, 2020). Although building computational models of individual intelligent human behaviors is useful, hand-engineered separate models fail to account for the generality and flexibility of human intelligence. It is important to go beyond matching human behavior on isolated tasks, and show how a single computational system could account for human intelligence across domains. A complete picture of human intelligence would have to show how the same computational system could learn new concepts, make jokes, give advice to a friend, make plans for the future, etc. If CTM is true, then such a computational system would be valuable in a deeper sense, beyond just as a way to make quantifiable predictions to test our theories of intelligence.

### 1.2.2 Programs as mental representations

Fodor (1975) argues that human thinking occurs in a mental language, a Language Of Thought (LOT), consisting of primitives (similar to words) that are composed with each other to form sentences. This can explain the productivity of thought, the seeming ability humans have to entertain potentially infinitely many (different) thoughts. Programs make good candidates for representations of mental primitives because they share the same language-like compositionality than enables this productivity. The Probabilistic Language Of Thought hypothesis (PLOT) (Goodman et al., 2014) postulates stochastic functions (programs) as its primitives. The composition of these stochastic functions defines a generative process over possible world states which when combined with the Bayesian statistics machinery can be used to learn from data through probabilistic inference. Motivated thinking is then analogous to sampling-based inference in this generative model of programs. These probabilistic programs can be thought of as a richer version of Bayesian graphical models, which have had success in modelling cognition as approximate Bayesian inference. PLOT combines the benefits of an LOT with those of Bayesian inference, which makes it a very promising computational architecture for modelling cognition and human intelligence.

One potential problem of the above approach is that for many probabilistic programs, exact Bayesian inference is intractable. Additionally, although there is a empirical evidence of such models matching human behavior (Lake et al., 2015), they rely on hand-written generative processes and so how these are acquired and change over time remains to be explained. Learning new generative processes and new concepts in PLOT corresponds to probabilistic program induction. It follows that to address both these issues and build even better models of human thinking and intelligence, we need to improve program synthesis performance.

## 1.3 Building machines with human-like intelligence

Human intelligent behavior has been a fruitful source of inspiration for building intelligent systems. Many benchmarks today evaluate artificial intelligent systems by comparing their performance to human performance. More fundamentally, the very concept of intelligence, and thus how artificial intelligent systems are evaluated and what is considered progress in artificial intelligence, is inextricably linked to human intelligence. It follows that computational models with symbolic programs as their representations and program synthesis as their learning mechanism should be useful for AI too. In fact they directly address some of the challenges of Artificial Neural Networks (ANN) which are currently at the forefront of the field. For example, unlike neural networks, but similar to people, program induction systems can successfully learn functions from just a few examples. This is because unlike neural networks, programs induction systems have strong inductive biases (strong priors), similar to the developmental "start-up software" which enables and guides human learning. For a more detailed treatment of other benefits of program synthesis models compared to ANN's see Lake et al. (2017).

Although cognitive science models with program representations that learn through program synthesis may be more human-like that does not necessarily imply that they are the best path towards building intelligent machines. It is possible that they are a lot harder to engineer than systems with neural representations. However we argue, that even if that is the case, we should strive to build AI systems that are *like* humans. If we are going to build machines that are useful and that we can trust, we must be able to understand them. One way to build machines we understand, is to make them more like us.

## 2 Properties introduction

| c001 | c010 | c077 |
|---|---|---|
| $f([2,4,3,2]) = [3]$ | $f([2,3,2,2,7,6]) = [3,2]$ | $f([1,7,2,0]) = [4]$ |
| $f([9,6,9,8,6]) = [9]$ | $f([3,9,8,6,5,1]) = [9,8,6]$ | $f([8,6,6]) = [3]$ |
| $f([0,0,0,0,0]) = [0]$ | $f([1,2,4,5,0,8,9,7,8]) = [2]$ | $f([2]) = [1]$ |
| $f([8,1,8]) = [8]$ | $f([5,5,5,1,1,5]) = [5,5,1,1,5]$ | $f([8,3,9,5,7]) = [5]$ |
| $f([5,7,5,7,5]) = [5]$ | $f([0,2]) = []$ | $f([]) = [0]$ |
| $f([1,1,1,1]) = [1]$ | $f([9,7,0,9,7,9,9,4,0,7]) = [7,0,9,7,9,9,4,0,7]$ | $f([4,4]) = [2]$ |
| $f([7,9,3,4]) = [3]$ | $f([3,8,3,3,8,3,3]) = [8,3,3]$ | $f([1,3,1,3,1,3]) = [6]$ |
| $f([7,7,7]) = [7]$ | $f([6,2,6,1,3,6,2,9]) = [2,6,1,3,6,2]$ | $f([7,9,9,9]) = [4]$ |
| $f$: ($\lambda x$ (singleton (third $x$))) | $f$: ($\lambda x$ (take (first $x$) (drop 1 $x$))) | $f$: ($\lambda x$ (singleton (length $x$))) |

Figure 1: Example concepts from Rule (2020)

Consider task c001 in Figure 1. We may not immediately realize what the hidden function $f$. However we may notice that the output always has length 1. This property "output has length 1" provides useful information for finding what $f$ is. Specifically it restricts the space of possible programs we need to search in, since we only need to consider programs that applied to these inputs result in outputs of length 1. Similarly for c010 we might notice the property "the output is a substring of the input" or more generally "the output is a subset of the input" which immediately restrict the programs we consider (e.g. we can disregard programs that add arbitrary constants to the output).[2] In fact, for concept c077 the properties "output has length 1" and "head of output is equal to the length of the input" fully specify its semantics. The conjunction of these two properties can be thought of as a logical specification of the program, where a program is correct if and only if these two properties are true of its corresponding input-output examples. It can also be thought of as a natural program,

[2]We cannot be certain that any of these properties is true of the latent program (i.e. true for all possible inputs to the latent program). However the more examples we observe for all of which a given property is true, the more suspicious it would be if it was not a true property of the latent program. We believe this has connections to the size principle (Tenenbaum, 1999), where more specific meanings become exponentially more likely with number of examples observed. All the properties discussed above are in fact true of the corresponding latent programs (shown in Figure 1) See appendix A.1 for a more detailed discussion on this.

a set of instructions that can be reliably conveyed to people to produce the intended output (see Acquaviva et al. (2021) who introduce the term for more discussion on natural programs). In this work we formalize these properties as symbolic functions of the spec (the input-output examples) and claim that they are useful for inductive synthesis. Specifically we make the following contributions:

- We introduce a novel way to score properties for a given spec and discuss its connections to Bayesian surprise (Itti and Baldi, 2009).
- We introduce *PropsimFit*, a novel online synthesis algorithm that uses properties during program search and show that it outperforms naive non-property baselines in the Rule (2020) list function inductive synthesis dataset.
- We discuss future ways to use properties for synthesis based on the insights gained from *PropsimFit* and its limitations.

## 3 Related Work

In the past the focus of program synthesis has been to synthesize programs from complete logical specifications (Manna and Waldinger, 1980). In what is often referred to as deductive synthesis, axioms and deductive rules transform the logical specification into a program. The process of transformation constitutes a proof that the program satisfies the logical specification. These systems require a lot of manual domain-specific work and are restricted in the types of programs they can construct. Additionally they require complete logical specifications which are often difficult to acquire.

Inductive synthesis algorithms take in inductive specifications, commonly input-output examples which are a lot easier to provide than complete logical specifications. Approaches like FlashFill (Gulwani, 2011) that fall under the FlashMeta framework (Polozov and Gulwani, 2015), take inspiration from deductive synthesis and use deductive rules to reduce the synthesis problem into smaller synthesis sub-problems.

Importantly these deductive rules rely on hand-written functions for each DSL primitive that essentially describe the inverse semantics of each DSL primitive. They employ these deductive rules to restrict the space of programs during enumerative search. The above make these systems a lot more generally applicable than deductive synthesis systems.

There is also a lot of recent work using neural networks to guide enumerative program search. In Deepcoder (Balog et al., 2016), the input-output examples (spec) are encoded neurally and a neural network is trained to predict the uses of DSL primitives. In Odena and Sutton (2020) the input-output examples are represented with a property signature, a list of program property values of the spec, which a neural network takes as input to predict DSL primitives. In some sense, neural networks can be used to automatically capture some of the expensive domain-specific inverse semantics that are manually specified in the deductive inference approaches. Having said that, the process of inference using neural-guided enumeration as performed above is qualitatively different from the deductive inference that systems that fall under the FlashMeta framework do. We develop synthesis algorithms with automatically generated properties that, like neural network approaches, require less manual domain-specific work, but unlike neural network approaches, capture and utilize hard constraints of the spec as in deductive inference systems.

## 4 Background

We rely on the formulation of properties proposed in Odena and Sutton (2020) and for convenience use the same notation. Let program $f :: t_{in} \rightarrow \tau_{out}$ and property $p :: (\tau_{in}, \tau_{out}) \rightarrow \mathit{bool}$ where $\tau_{in}$ and $\tau_{out}$ are the input and output types respectively. Then if $S$ is a set of $\tau_{in}$ values, let $p(f, S) = \{p(x, f(x)) \mid x \in S\}$. Then because $p(f, S)$ is a set of boolean values it will either be $\{\text{True}\}$, $\{\text{False}\}$ or $\{\text{True}, \text{False}\}$. To simplify notation, let $\Pi(\{\text{True}\}) = \mathit{AllTrue}$, $\Pi(\{\text{False}\}) = \mathit{AllFalse}$ and $\Pi(\{\text{True}, \text{False}\}) =$

$Mixed$ [3] . Finally let $V(f)$ be the (potentially infinite) set of valid inputs for program $f$. We can now talk about the property value $\Pi\left(p(f, V(f))\right)$ of a given program $f$ and property $p$. Odena and Sutton (2020) define the property signature $\mathrm{sig}(P, f)$ of a program $f$ and a property sequence $P$ as,

$$\mathrm{sig}(P, f)[i] = \Pi(p_i(V(f)))$$

In programming by example, we only observe a finite set of input-output pairs $S_{io}$ with the goal of inferring $f$ that generated them. Odena and Sutton (2020) use this set $S_{io}$ to compute the estimate property signature of $f$,

$$\widehat{\mathrm{sig}}(P, f)[i] = \mathrm{iosig}(P, S_{io})[i] = \Pi(\{p_i(x_{in}, x_{out}) \mid (x_{in}, x_{out}) \in S_{io}\})$$

We can say the following about the estimate $\widehat{\mathrm{sig}}(P, f)[i]$. If $\widehat{\mathrm{sig}}(P, f)[i] = \mathrm{Mixed}$ then also $\mathrm{sig}(P, f)[i] = \mathrm{Mixed}$. If $\widehat{\mathrm{sig}}(P, f)[i] = \mathrm{AllTrue}$ then $\mathrm{sig}(P, f)[i]$ could be either AllTrue or Mixed. Similarly if $\widehat{\mathrm{sig}}(P, f)[i] = \mathrm{AllFalse}$, $\mathrm{sig}(P, f)[i]$ could be either AllFalse or Mixed. See Appendix A.1 for a more detailed treatment of estimated property values.

Odena and Sutton (2020) use this estimated property signature to featurize the spec and condition program search on it using a neural network to predict the uses of the DSL primitives. In the follow-up work BUSTLE (Odena et al., 2020), they use a neural network in the loop during bottom-up search to predict how likely intermediate sub-expressions are to appear in the final program, where sub-expressions are executed and featurized using property signatures. Although they effectively demonstrate that the proposed property-based systems outperform non-property baselines in the examined synthesis domains, their synthesis systems are unsatisfying in the

[3] We note that properties can be generalized to $p :: (\tau_{in}, \tau_{out}) \rightarrow \tau_*$ where $\tau_*$ is any type in the DSL. Although the set of of possible values for $p(f, S)$ would not be as small, they could always be summarized with AllSame when $|p(f, S)| = 1$ and Mixed when $|p(f, S)| > 1$. This an interesting direction for future work as it might facilitate automatically generating properties and picking out useful constants from the spec.

following ways:

- **They do not explicitly utilize the precise constraints implied by properties**. Recall the examples in the introduction, and how properties provide precise constraints regarding the set of possible correct programs. We want to build a system that makes these constraints explicit and a central part of how it uses properties during search, rather than as features for a neural network to consume.

- **They are not as good cognitive science models**. There are tasks that are neither immediately obvious nor impossible us to solve, for which it feels that we engage in some *online* inference process that does not seem to rely on having seen thousands of input-output examples, their corresponding properties, and what programs they correspond to. We claim that properties play an essential role in this process and wish to build a computational model that uses properties to describe it. Additionally, regardless of whether one believes neural networks can learn to approximate this inference process, they are not interpretable and so can only give us limited insight into this process and the inference steps people use to arrive at the final programs.

## 5 Approach

We want to build a model that matches the following intuitions about how people use properties to solve inductive synthesis tasks:

a) People notice and pay attention to program-like properties of the spec.

b) People use these properties to restrict the space of possible programs they need to search in. From a bayesian inference perspective, we would say people use properties of the spec to refine their approximate posterior distribution over programs.

As a first step towards a), in section 5.1 we present a way to score properties for a given spec and discuss its connections to Bayesian surprise (Itti and Baldi, 2009). As a first step towards b), in section 5.2 we present *PropsimFit*, a synthesis algorithm that uses properties to improve its distribution over programs.

## 5.1 Scoring properties

In the experiments described below we generate properties by enumerating from the program DSL (see section 7.4.2 for ways to improve this). This results in a list of thousands of properties. In order to better match our intuition that people notice and use a significantly smaller number of properties (as well as for computational efficiency considerations) we want to keep a small set of the most useful properties for a given spec[4]. To accomplish this we need a way to score properties of programs. Intuitively we assign a high score to rare property values and a low score to more common ones (under our prior distributions over programs). The value AllTrue for property "output starts with [4,4,7,8]" is rare and gets assigned a high score, whereas the value AllTrue for property "output has length > -2" is true for any program we consider and so get assigned the lowest possible score.

Assuming we have a library L which defines a prior over programs $\mathbb{P}(f \mid L)$ we score a property $p$ for a given program $f$ based on how surprising the property value $c = \Pi\left(p(f, V(f))\right)$ is under the prior over programs where $c \in \{\text{AllTrue}, \text{AllFalse}, \text{Mixed}\}$. The score is thus a function of only $p$ and $c$,

$$\begin{aligned}\text{score}(p, c) &= -\log\left[\mathbb{P}\left(\Pi\left(p\left(\mathbf{f}, V(f)\right)\right) = c \mid L\right)\right] \\ &= -\log\left[\sum_f \mathbb{P}\left(\Pi\left(p\left(f, V(f)\right)\right) = c \mid f\right) \cdot \mathbb{P}(f \mid L)\right]\end{aligned}$$

For a given spec, we can then score each of the properties based on the estimated property value $\Pi(p(f, S_i))$. In Table 1 we see the highest scoring properties for task c001 where we equally weight the primitives of L and enumerate until we get 10,000

[4]Note in Odena and Sutton (2020) thousands of properties are used to make up the property signature; it is less of an issue in their system since the neural network can learn to weight them appropriately.

programs to estimate the prior over programs $\mathcal{P}(p \mid L)$.

| property | score | $\Pi(p(f, S_i))$ | $\Pi(p(f, V(f)))$ |
|---|---|---|---|
| output element at index 0 equal input element at index 2 | 4.25 | AllTrue | AllTrue |
| output has length 1 | 2.14 | AllTrue | AllTrue |
| output is shorter than input | 0.98 | AllTrue | AllTrue |
| output contains input element at index 2 | 0.93 | AllTrue | AllTrue |
| output contains number 2 | 0.91 | AllFalse | Mixed |
| output contains number 4 | 0.89 | AllFalse | Mixed |
| output has length 4 | 0.88 | AllFalse | AllFalse |
| output contains number 6 | 0.86 | AllFalse | Mixed |
| output has length 3 | 0.78 | AllFalse | AllFalse |
| output is same length as input | 0.69 | AllFalse | AllFalse |

Table 1: Highest scoring handwritten properties for task c001 (see Figure 1 for reference) whose program $f$ is "remove all but element at index 2". The first column is a short description of the property, the second column is its score computed using the estimated property value, the third column is the estimated property value based on the 8 observed input-output examples generated from applying $f$ to inputs $S_i$, and the fourth column is the true property value for latent program f (computed over all valid inputs $V(f)$).

The resulting highest scoring properties "output element at index 0 equal input element at index 2" and "output has length 1" seem to match our intuitions about which properties relate the most to the semantics of the program. The next two properties "output is shorter than input" and "output contains input element at index 2" are also always true of the latent program but they are not as specific. Of the remaining properties some are incorrectly estimated to be AllFalse when they should be Mixed and others are correctly estimated but not as indicative of the semantics of the program e.g. "the output has length 3" is AllFalse i.e. "the output never has length 3". For more discussion on estimated property values see Appendix A.1.

### 5.1.1 Connection to bayesian surprise

Another way to think of the score is that it picks out the most informative properties. A high-scoring property is only true for a small fraction of the programs under consideration and so allows us to exclude a large fraction of the programs. Under the bayesian view, the highest scoring property values are the ones that are the least

| property | score | $\Pi(p(f, S_i))$ | $\Pi(p(f, V(f)))$ |
|---|---|---|---|
| output has length 1 | 3.12 | AllTrue | AllTrue |
| all output elements are less than 0 | 1.66 | AllFalse | AllFalse |
| output list has length 0 | 1.64 | AllFalse | AllFalse |
| output list has length 2 | 1.46 | AllFalse | AllFalse |
| all output elements are less than 7 | 1.32 | AllTrue | Mixed |
| all output elements are less than 8 | 1.22 | AllTrue | Mixed |
| output list has length 3 | 1.17 | AllFalse | AllFalse |
| output list has length 4 | 1.12 | AllFalse | AllFalse |
| all output elements are less than 9 | 1.22 | AllTrue | Mixed |
| output contains number 7 | 0.94 | AllFalse | Mixed |

Table 2: Highest scoring handwritten properties for task c077 (see Figure 1 for reference).

likely under the prior and so updating the prior over programs to match the new estimated property value posterior would result in a large update in our beliefs. In fact the score as defined above is exactly the bayesian surprise (Itti and Baldi, 2009) of the estimated property value $\Pi(p(\mathbf{f}, S_i))$, which is a measure of the difference between its posterior and prior distribution. Formally, the property score is the KL divergence between the estimated property value posterior after conditioning on the observed spec $S_{io}$ and the estimated property value prior. The estimated property value prior is given by $\mathbb{P}\left(\Pi\left(p\left(\mathbf{f}, S_i\right)\right) \mid L\right)$ and the estimated property value posterior is given by $\mathbb{P}\left(\Pi\left(p\left(f, S_i\right)\right) \mid L, S_{io}\right)$ which is deterministic (i.e. equal to one of $\{\text{AllTrue}, \text{AllFalse}, \text{Mixed}\}$ with probability 1).

It would be interesting to investigate whether the property value score tracks which properties people notice and pay attention to when looking at the spec in the same way that Bayesian surprise tracks human attention in Itti and Baldi (2009).

#### 5.1.2 Limitations

- We have been claiming that the property score should pick out the most *useful* properties but to show that they are *useful* we must show that they improve synthesis performance compared to ablations that don't use properties.

- Since we have not run any human experiments we cannot draw any conclusions as to whether this score captures the properties people notice and use to solve tasks. In fact we suspect that what properties people notice also depends on how visible they are (see discussion on visibility in Rule (2020)) which is not captured by the current formulation of the score. We conjecture that the property score most likely tracks the properties people find the most useful/suspicious (and possibly the ones people pay the most attention to *if noticed*) rather than the ones that are most noticeable. Consider that for any task (with latent program $f$) the highest scoring property will always be "f executed on the input equals the output" but people that are unable to solve the task never notice this property. However if they did notice, it would by definition be the most useful property for program synthesis.

## 5.2 PropsimFit

*PropsimFit* provides a way to alter the prior distribution over programs to a distribution over programs under which programs that share many properties with the latent program are more probable.[5] *PropsimFit* scores programs from the original prior distribution of programs based on how many properties they share with the latent program, and fits a Probabilistic Context Free Grammar (PCFG) to the highest scoring of these to get a new approximate posterior distribution of programs.

### Propsim score

To score two programs based on how many properties they share with each other we propose the *propsim* score. For a set of properties $P$, library $L$ and a prior over

[5]Note that by "share many properties" here and later on we really mean share property values e.g. the latent program has property value AllTrue for property "output same length as input" and so do most samples from the posterior distribution of programs.

programs $\mathbb{P}(f \mid L)$, we calculate the similarity of programs $f_a$ and $f_b$ as follows,

$$\text{propsim}(f_a, f_b) = \sum_i \begin{cases} \text{score}(p, c) & \text{if } \text{sig}(P, f_a)[i] = \text{sig}(P, f_b)[i] = c \\ 0 & \text{otherwise} \end{cases}$$

Scored this way, not all properties count equal. The higher the score of the property the two programs share i.e. the more rare (under the prior over programs) the property they share is, the higher its contribution to their propsim score. One potential downside of this scoring method is that each property independently (of other properties) contributes to the propsim score which may not be desirable for properties that are very similar to each other.

The propsim score can also be used to score the similarity between two specs, or between a program and a spec by replacing $\text{sig}(P, f_b)$ above, with the estimated property signature $\widehat{\text{sig}}(P, f_b) = \text{iosig}(P, S_{io})$.

**PropsimFit Algorithm**

Intuitively in *PropsimFit* we modify the program grammar to generate programs that have a high propsim score with the spec. The algorithm is as follows,

1. Handwrite or automatically generate set of properties $P$

2. Sample/enumerate from DSL to get set of programs $F = \{f \mid f \sim \mathbb{P}(f \mid L)\}$

3. For task $x \in X$ (where task x is a set of input-output examples) with inputs $S_i$

    (a) Create synthetic tasks $H = \{\text{makeTask}(f, S_i) \mid f \in F\}$ where $\text{makeTask}(f, S) = \{(x_{in}, f(x_{in})) \mid x_{in} \in S\}$

    (b) Compute $\text{propsim}(\text{iosig}(P, x), \text{iosig}(P, h)) \quad \forall h \in H$

    (c) Fit the PCFG to the $n$ most similar $h \in H$ (according to propsim score) i.e. set the weights of $L$ using the inside-outside algorithm so that the probability of the n most similar programs is maximized. This results in $G_{fitted}$ which instantiates the approximate posterior $\mathbb{Q}(f \mid L, x)$.

(d) Enumerate from grammar $G_{fitted}$ proposing programs $f$ in decreasing order of probability under $\mathbb{Q}(f \mid L, x)$ until timeout

# 6 Experiments

## 6.1 Preliminaries

**Dataset:** We use the dataset introduced by Rule 2020, which consists of list functions that transform list of integers to list of integers. For the experiment below we use the first 80 concepts, which only contain integers in the range 0 to 10. All concepts are manually generated to capture broad variation in how difficult they are for human learners and in the kinds of algorithmic reasoning they require. Each concept is specified by 11 input-output examples; we use the first 8 to induce a program and the remaining 3 to evaluate if the induced program is correct.

**DSL:** We use the rich DSL (see appendix A.4 for more details) from which the concepts were formed, the more basic DSL (see A.3) and the starting DSL used in the Dreamcoder list domain (see A.5).

**Properties:** We hand-write a set of properties of the spec that seem intuitively useful when manually solving the 80 tasks, parametrizing them when possible. We follow the formalization of Odena et. al restricting every property to type $(\tau_{in}, \tau_{out}) \rightarrow$ *bool*. See appendix A.2 for a full list of the handwritten properties. We also experiment with automatically generating properties by enumerating from the rich DSL for programs of type $(\tau_{in}, \tau_{out}) \rightarrow$ *bool* keeping only ones that result in a unique signature over the 80 tasks.

**PropsimFit Details:** We enumerate from the prior $\mathbb{P}(f \mid L)$ where all primitives in L are equally weighted until we get 10,000 programs that make up $F$. We run the inside-outside algorithm on the 50 most similar tasks to get a task-specific PCFG from which we enumerate for 600 seconds for each task.

**Baselines:** All models including *PropsimFit* result in a PCFG for each task from which we perform type-directed enumeration in order of decreasing program probability. The baselines differ in how they set the probabilities of the grammar productions. The simplest baseline *Uniform* equally weights all productions. The *AllFit* baseline assigns grammar production probabilities using the inside-outside algorithm fitting to *all* synthetic tasks $H_x$[6] (unlike *PropsimFit* which fits on the 50 most similar tasks of $H_x$). The *Neural* baseline is the neural recognition model from Ellis et al. (2021) which consists of a GRU, a type of recurrent neural network, that encodes the input-output examples into a vector which is passed into an MLP layer to predict the PCFG probabilities. As in the list domain experiments in Ellis et al. (2021) it is trained for 10,000 gradient steps.

## 6.2 Results

| | Number of tasks (out of 80) solved with fixed enumeration time | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Rule rich DSL | | | | Rule basic DSL | | | | DC list DSL | | | |
| | 10s | 25s | 100s | 600s | 10s | 25s | 100s | 600s | 10s | 25s | 100s | 600s |
| PropsimFit (handwritten) | 4 | 28 | **53** | **56** | 8 | **17** | **19** | 19 | 8 | **26** | **28** | **29** |
| PropsimFit (automatic) | 6 | 24 | 49 | 55 | **10** | **17** | 18 | **20** | **9** | 21 | 22 | 22 |
| Neural | **24** | **40** | 48 | 53 | 9 | 15 | 16 | 16 | 8 | 22 | 25 | 28 |
| AllFit | 1 | 23 | 44 | 52 | 9 | 14 | 15 | 17 | 6 | 19 | 26 | 28 |
| Uniform | 1 | 3 | 25 | 32 | 10 | 13 | 17 | 17 | 6 | 14 | 16 | 16 |

Table 3: Number of tasks solved for different single CPU enumeration times in Rule (2020) list function domain using the rich DSL. Each column represents a vertical slice from Figure 2.

[6]This does not result in the same grammar as *Uniform* because the 10,000 programs are the highest probability programs rather than samples from *Uniform* and because we exclude programs that result in the same output for all examples.

### 6.2.1 Symbolic properties are useful for program search in Rule (2020)

After enumerating for 600 seconds, *PropsimFit* solves more tasks than all baselines for all three DSLs. In the Rule (2020) rich DSL and the Rule (2020) basic DSL experiments, *PropsimFit* solves 3 more tasks than the next best baseline, while in the DC list DSL experiments, *PropsimFit* solves 1 more task than the next best baseline. Generally *PropsimFit* with handwritten properties seems to perform marginally better than with automatically generated properties. Note that 1. *PropsimFit* with handwritten properties outperforms *AllFit* for all three DSLs and 2. *AllFit* is a form of ablation of *PropsimFit* that excludes property information. Taken together these indicate that properties capture information that is useful for program search.

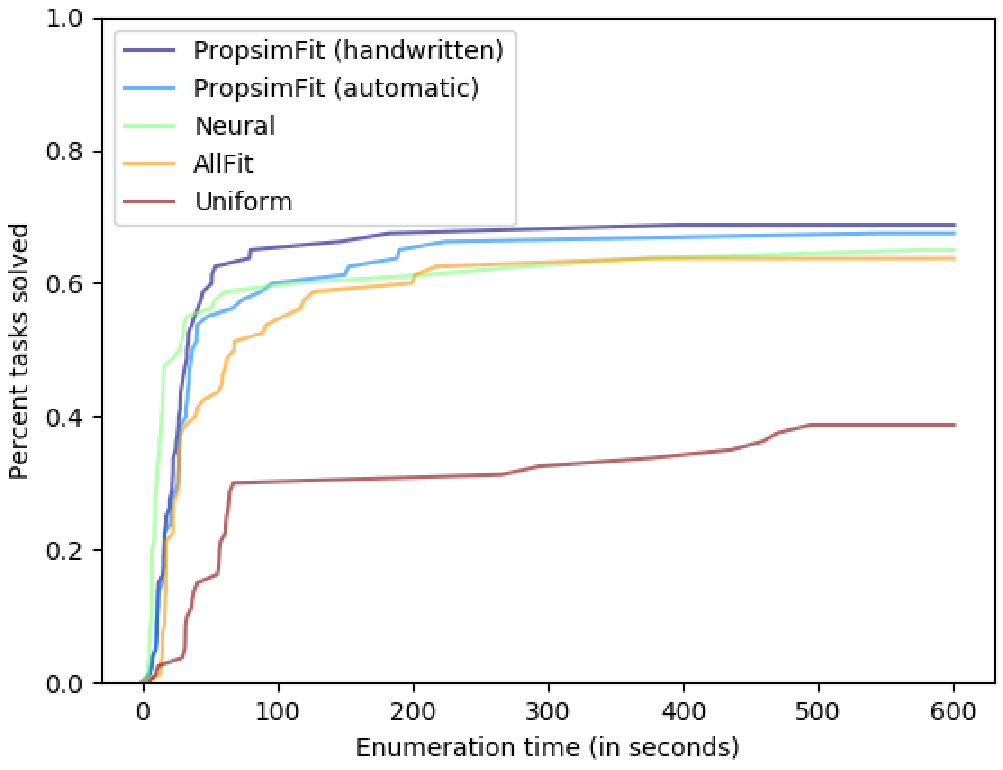
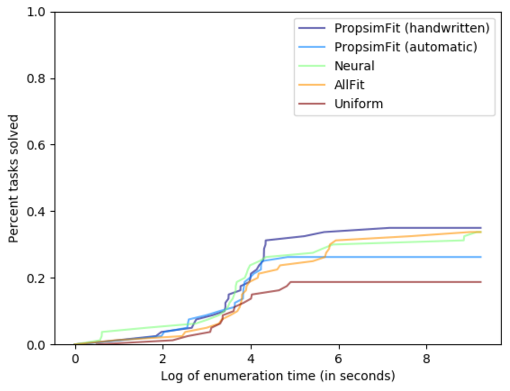
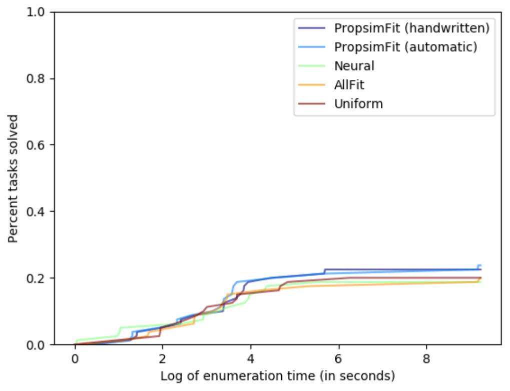


Figure 2: Cumulative percent tasks solved over time in Rule (2020) list function domain using the rich DSL (see A.3 for DSL details).



(a) Results using starting DSL from Ellis et al. (2021) list function domain



(b) Results using basic DSL from Rule (2020) (see A.3)

Figure 3: Cumulative percent tasks solved over time in Rule (2020) list function domain with other DSLs.

### 6.2.2 Following property signal improves approximate posterior distribution

For the vast majority of tasks solved, the approximate posterior probability of the best program found is higher under *PropsimFit* than under *AllFit* (see Figure 4 and Figure 5). We compare to *AllFit* instead of *Uniform* because we want to specifically show the contribution of properties (rather than the effects of fitting on programs from type-restricted enumeration that passed the filtering step). It is worth noting that for some of the easier tasks, the task solution is included in the 50 most similar tasks found according to the propsim score and so it is less surprising that the ground truth program has a high probability under the *PropsimFit* grammar. However, for the remaining tasks, the fact that the probability of the ground truth programs is higher under the *PropsimFit* grammar compared to the *AllFit* baseline indicates that programs that are similar under the propsim score are more likely to have common primitives and are thus a good signal to follow.

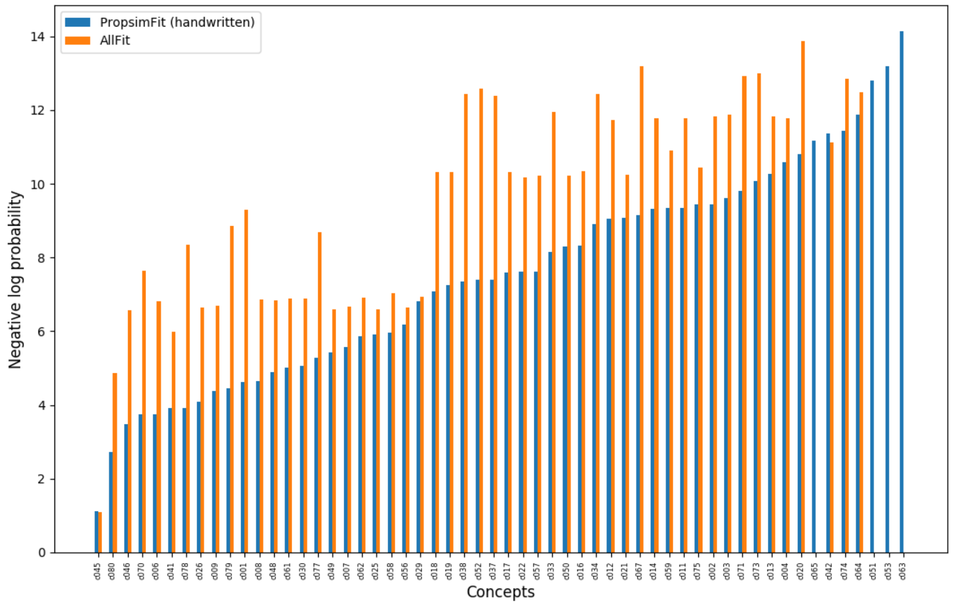
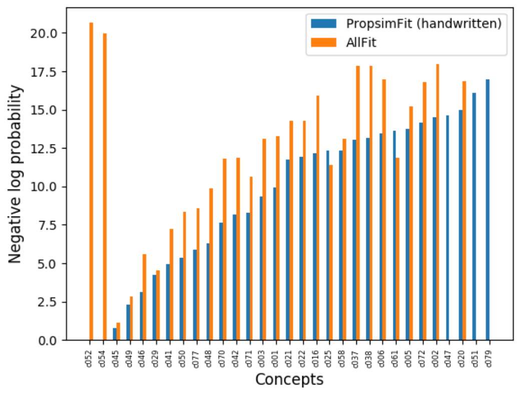
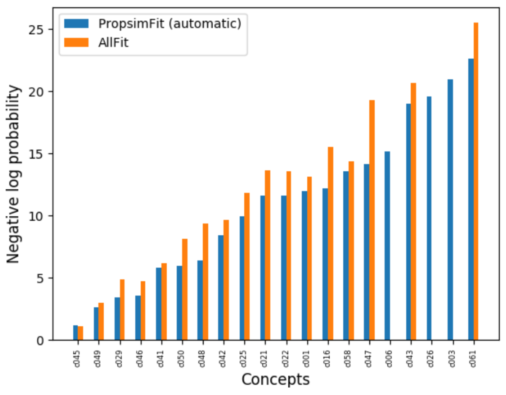


Figure 4: Approximate posterior of MAP program discovered for each concept (concepts for which no model discovered a program are excluded) in Rule (2020).



(a) Results using starting DSL from Ellis et al. (2021) list function domain



(b) Results using basic DSL from Rule (2020) (see A.3)

Figure 5: Approximate posterior of MAP program discovered for each concept (concepts for which no model discovered a program are excluded) in Rule (2020) list function dataset, comparing the baseline AllFit with the best performing PropsimFit model.

The experiments support the insight that properties are useful for program search and that improving the program prior using shared properties with the spec as a signal results in a distribution for which the ground truth program is more probable in the Rule (2020) list function domain. To what extent these insights generalize to other domains remains to be tested.

# 7 Looking forward

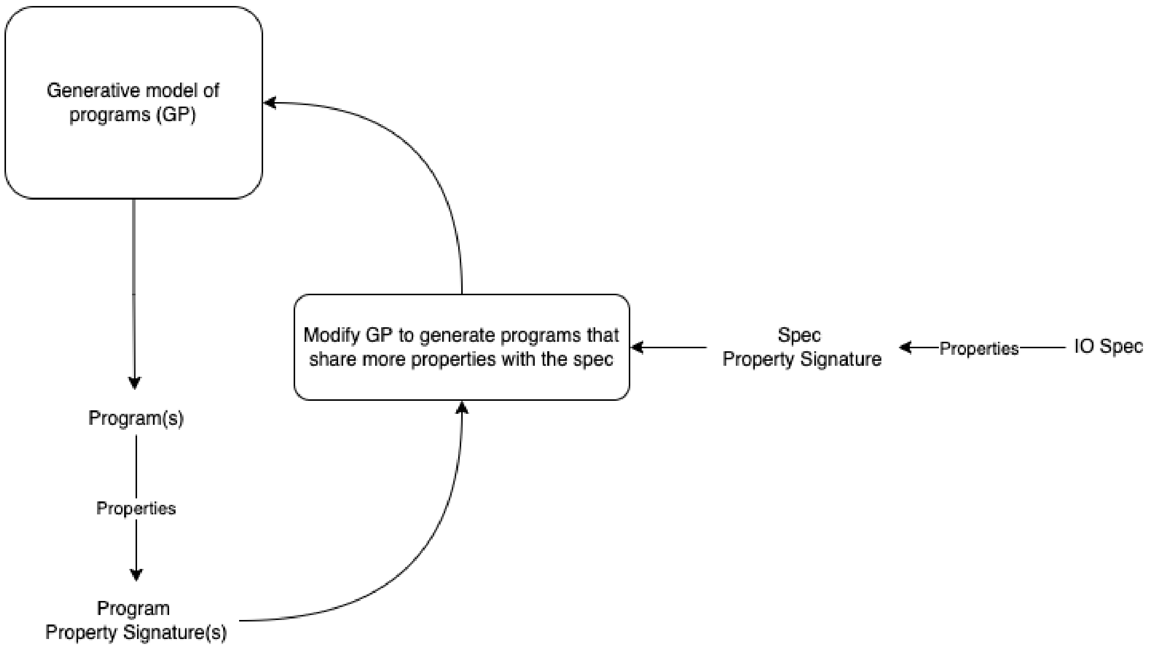

Generative model of programs (GP)
Modify GP to generate programs that share more properties with the spec
Spec Property Signature
Properties
IO Spec
Program(s)
Properties
Program Property Signature(s)

Figure 6: Diagram of general approach for using properties for program synthesis. *PropsimFit* is a special case of this.

*PropsimFit* modifies a generative model of programs following the signal of shared properties. The generative model is modified so that it generates programs that share more properties with the spec to solve. This approach relies on the assumption that a generative model of programs that share more properties with the spec will also be a better model of the latent program (i.e. generate it with high probability). In Section 6.2 we show *PropsimFit* solves more task than the naive baselines in Rule (2020), and results in better approximate posterior distributions of the latent programs. Although there there is still much room for improvement, we believe these results indicate that there is merit to the more general approach (discussed above and illustrated in Figure 6). In *PropsimFit* the generative model of programs is a PCFG combined with type-directed top-down enumeration in order of decreasing program probability. It is modified by changing the grammar production probabilities to maximize the probability of the programs most similar to the spec according to the propsim score. Below we discuss some of the limitations of these design decisions and refer to concrete ways to address them:

1. **Extract more information from similar programs**: In *PropsimFit* the only information we can extract from similar programs is about the marginal probabilities of individual primitives. If two primitives always co-occur in similar programs but never occur individually, fitting a PCFG using the inside-outside algorithm to these programs will not capture this. How much information we can extract from a list of similar programs and how we generalize from it depends both on the kind of generative model of programs we use and the ways in which we modify it.

2. **Explore different ways to follow shared property signal to improve generative model of programs**: In *PropsimFit* we update the generative model of programs based on the propsim score computed from all properties of the spec. *PropsimFit* follows the signal of shared properties to improve its generative model of programs. There are many more options to explore here regarding how to follow the signal of shared properties. We could chose to follow the signal from any subset of the properties (ranging anywhere from a single property to all of the properties of the spec). If the generative model is modified iteratively, we could dynamically change how many and which property signals we choose to follow at each iteration. We could even cast this as a reinforcement learning problem and learn these decisions based on whether they lead to tasks solutions. We conjecture that the property score introduced in Section 5.1 could be useful for making these decisions.

3. **Improve automatically generated properties**: The general approach relies on the fact that it is easier to generate useful properties than it is to directly generate the latent program. Both in this work and in Odena and Sutton (2020), Odena et al. (2020) properties are naively generated using the program DSL. Although generating properties this way is sufficient for outperforming the respective synthesis baselines, we believe it is crucial to improve the way properties are generated to ensure that they capture the semantics of the latent program. This is an essential feature that upper-bounds the performance of any

of property-based synthesis methods. Even if we could produce a generative model that only outputs programs with identical property signatures as the spec, it would only be useful if properties that made up the property signature captured a significant part of the semantics of the latent program.

In Section 7.1 we discuss extensions to *PropsimFit* that address 1. and 2. In Section 7.2 we present preliminary experiments combining properties with MCMC and discuss future directions and how these relate to 1. and 2.. In Section 7.3 we propose a different generative model of programs consisting of a population of partial programs and discuss its connection to 1. and 2. All the above synthesis systems would benefit from better automatically generated properties and a DSL more tailored to the datasets to be solved. In Section 7.4 we discuss how to achieve both of these by combining property-based synthesis algorithms with Dreamcoder library learning.

## 7.1 PropsimFit extensions

As discussed above *PropsimFit* is limited in the kind of information it can extract from similar programs. One straightforward way to improve *PropsimFit* is to replace the unigram grammar with a bigram grammar. From a list of similar programs we would again use the inside-outside algorithm to learn the bigram probabilities. This change should enable richer learned generative models that more tightly model the distribution of similar programs. Additionally, we can extract more information from the list of similar programs, by using the compression algorithm from Ellis et al. (2021) to invent new primitives from subprograms that appear in multiple similar programs. In this way, *PropsimFit* is no longer limited to only being able to learn unigram or bigram probabilities from the list of similar programs, but rather can extract and upweight any arbitrary skip-gram that is repeated in multiple similar programs.

In conjunction with the above, it is worth exploring fitting grammars following the signal from either one property or a subset of properties instead of using the propsim score which considers all properties of the spec. For example, consider a spec that only has two properties that are AllTrue. We want to modify our original generative model so that it produces programs that share these two properties i.e. also have value AllTrue. To accomplish this we could run the *PropsimFit* algorithm, either once or iteratively. Alternatively we could try to first get a good generative model of programs that share one of the two properties and fit a grammar to the subset of programs that share that property (using the inside-outside algorithm and the improvements described above). We could then use the resulting grammar as the generative model to get a new list of programs fitting to the subset of these that share the second spec property. The resulting grammar should now generate more programs that share both properties and could be more effective at this than a grammar directly fitted to a list of programs (generated from the original grammar) that share both properties. Of course without having run experiments the usefulness of the above is limited, but it serves as a concrete example of how to modify the generative model of programs to *incrementally* share more spec properties.

## 7.2 Properties and MCMC

### Overview

We also use the propsim score to guide stochastic search in Fleet (Piantadosi, 2020). In Fleet synthesis is framed as sampling from a Bayesian posterior where the learner observes a set of input-output examples, and must infer the most likely latent program to have generated these. The prior is defined by the grammar and the likelihood typically specifies that the output is observed with some noise. Such approximate likelihoods result in more programs with non-zero posterior probabilities than an exact likelihood that assigns 1 to programs which satisfy the spec and 0 to all the rest. When used with algorithms like MCMC this can allow for more efficient exploration of the space of programs. Below we show results from experiments using the propsim score as an approximate likelihood in Fleet.

## Preliminary Experiment

We use the first 100 concepts from Rule (2020) and the same basic DSL used in model experiments in Rule (2020). Appendix A.3 has the the full list of every primitive name, its type, and a short description of what it does as taken from A.3. Note for these experiments we only use integers from 0 to 10 (inclusive).

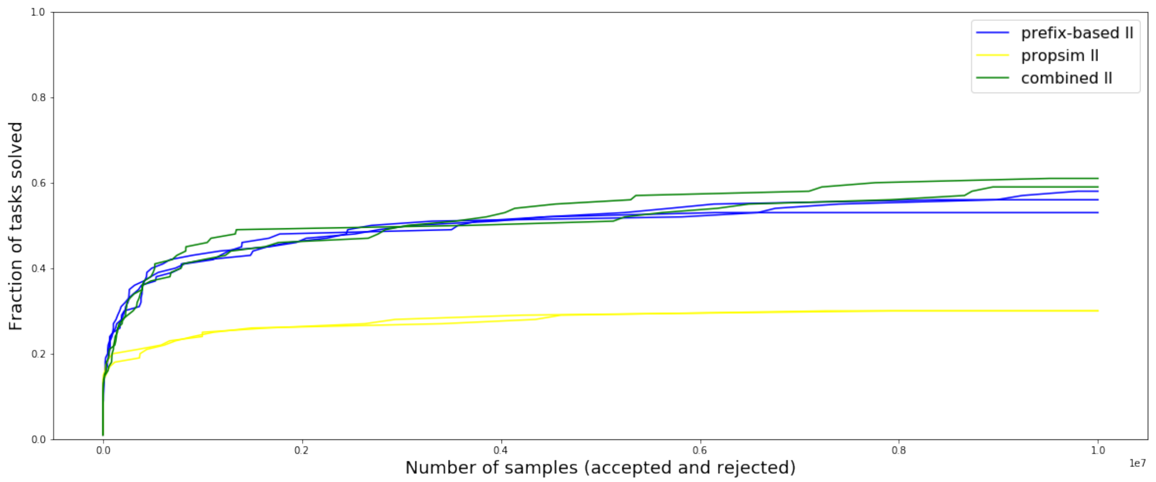


Figure 7: Cumulative percent of Rule (2020) tasks solved (for 100 tasks) as a function of MCMC samples using different approximate likelihoods.

Fleet with an approximate likelihood that is just the property score solves fewer tasks for the same number of samples than the existing prefix-based approximate likelihood. Combining the two by multiplying them together results in marginally more tasks solved compared to using the prefix-based approximate likelihood alone.

## Discussion

Although in these preliminary experiments were not able to conclusively show that a property-based approximate likelihood can replace the existing prefix-based one (see Figure 7), we are optimistic that incorporating properties in Fleet could further improve Fleet's already high performance in the Rule (2020) list function domain. We could experiment with more ways to combine the property score with the prefix-based score, filtering properties that are too similar to each other, as well as using a different set of (potentially automatically generated) properties. Alternatively, we could use properties to guide MCMC proposals. One way of doing this would be to generate proposals in the same way Fleet currently generates proposals by replacing candidate program subtress, but use properties to filter them only keeping proposals whose corresponding specs share as many or more properties than the candidate program. We could also have different MCMC chains use different approximate property-based likelihoods each corresponding to a different subset of properties. Combined with the existing parallel tempering algorithm (Vousden et al., 2016) which maintains many chains of different temperatures and periodically swaps them, this could be an effective way to explore the space of programs while prioritizing programs that share many properties with the spec to solve, while preventing "getting stuck" in local minima.

We can also use MCMC to replace enumeration in PropsimFit. For the prior distribution which is used in the proposal mechanism we would use the same unigram PCFG and for the approximate likelihood we would keep the default prefix-based likelihood (alternatively we could use any of the property-based proposal mechanism and approximate likelihoods described above). We would then run MCMC to generate a list of programs (samples) that we can score based on how many properties they share with the spec to solve, for example by using the propsim score. Depending on if properties are used during MCMC sampling this step might be unnecessary. Finally, we can fit a new grammar to the highest scoring programs (again using the inside-outside algorithm) from which can enumerate or sample with MCMC (using it as the new prior). Additionally, we could use the highest scoring programs as the seed programs MCMC starts with and restarts to when it "gets stuck". In the same way we can run *PropsimFit* iteratively, we can repeat this process iteratively where all the same considerations and strategies regarding how many and which properties to use at each iteration also apply.

## 7.3 Partial programs with holes

Consider the partial program "(filter input HOLE)" where HOLE can be replaced with any function that takes an integer and returns a boolean. Regardless of how this partial program is filled in, the final program will have value AllTrue for property "output length is <= input length". Learning this fact *automatically* could be really useful. If the spec we are solving shares this property this should make us more confident we are on the right track, whereas if it doesn't we should exclude this partial program and its potential continuations from consideration. We suggest that although we might not be able to prove this relationship to be true, we could sample many times from the hole and if all of the resulting programs share the property judge that it is highly likely to be true of all possible continuations.

Based on the above we believe it is worth exploring a generative model of programs that maintains a population of partial programs that are incrementally filled in so that they share more properties. A key benefit of this representation is that we can modify the generative model of programs (by extending the partial programs) in a way that monotonically increases the number of properties satisfied.

Another way to take advantage of the fact that some partial programs always share certain properties is to use Monte Carlo Tree Search for program synthesis (see Lim and Yoo (2016) details on how to set up) scoring programs using the propsim score. If the spec to solve has value AllTrue for property "output length is <= input length" any path down the tree that begins with the "filter" primitive would correspond to a program with a high propsim score and thus upweight the corresponding tree path. Effectively this upweights programs with "filter" as their outermost primitive.

## 7.4 Synergistic Dreamcoder library learning with properties

### 7.4.1 Learned concepts can give rise to better properties

Consider a task for which the program is "remove all the non-prime numbers from the input list". Also assume that through library learning we have learned the primitive "is_prime". With a neural recognition model, as in Dreamcoder (Ellis et al., 2021), even if the primitive "is_prime" is included in the program of one of the random

Helmholtz tasks we train on, it seems unlikely that the neural network could learn the general concept of primality, as neural networks are good at interpolating not extrapolating (it could at best learn to exclude all the prime numbers in the training set but not a general rule for deciding primality). However, by generating properties from the program DSL (which includes the new "is_prime" primitive) we can hope to generate a property like "does_output_contain_only_prime_numbers" which (we hypothesize) should be useful for inferring the program "remove all the non-prime numbers from the input list". Rather than training a neural network to find the features of a given task and from those predict the unigram/bigram probabilities of the DSL as in the recognition model (the last layer of the recognition model can be thought of as the feature layer), we can leverage the DSL concepts that we already know are relevant to many tasks (e.g. primality) using symbolic properties as features. As the DSL concepts become richer and increasingly specialized to our dataset of tasks we expect to benefit more from using properties (that take advantage of these concepts) for inference. This could result in a positive feedback loop where better properties help solve more tasks enabling more library learning which in turn could lead to even better properties.

### 7.4.2 Better automatically generated properties

The key argument from above is that learned concepts can be used to construct useful properties. How this happens depends on how we automatically generate properties.

In *PropsimFit* we automatically generate properties by sampling/enumerating from the program grammar. Combined with a property-based synthesis approach automatically generating properties this way could benefit from Dreamcoder library learning since at each iteration properties would be generated from a richer, more suitable DSL.

Although we expect properties to share primitives with the programs, it is possible that the program grammar which is updated to maximize the probability of the discovered programs is not well suited for generating properties. We might instead want to learn a dedicated property grammar. The initial primitives for this grammar could be handwritten (in the same way that we initially handwrite the primitives of the DSL) or they could be taken from the program DSL. The initial property grammar could then be improved by: **1.** Taking fragments from programs of solved tasks and including them as primitives in the program DSL. **2.** Running Dreamcoder compression on the set of property programs that were useful in solving tasks. How to decide whether a given property was useful depends on the property-based synthesis algorithm used. In *PropsimFit* and other methods that use the propsim score we can consider properties that contribute significantly to the propsim score as useful. In Odena and Sutton (2020) we could judge the usefulness of a property using standard neural network feature importance techniques. In a system like the one described in section 7.3 each task would use a small number of properties and so we could consider all of them useful. **3.** Adding learned primitives from the program DSL as the program library grows.

## 8 Conclusion

In this work we extended the property formulation by presenting a way to score properties for a given program or spec. We introduced *PropsimFit*, a novel property-based synthesis algorithm and showed that it outperforms naive baselines in the Rule (2020) list function domain. We argued that this indicates that there is merit to the general approach of modifying a generative model of programs following the signal of shared properties. Finally we outlined future improvements to *PropsimFit*, and explored different synthesis systems that share the same general approach but overcome some of *PropsimFit*'s limitations.

# A Appendix

## A.1 Estimating property values and suspicious coincidences

c001

$f([2,4,3,2]) = [3]$

$f([9,6,9,8,6]) = [9]$

$f([0,0,0,0,0]) = [0]$

$f([8,1,8]) = [8]$

$f([5,7,5,7,5]) = [5]$

$f([1,1,1,1]) = [1]$

$f([7,9,3,4]) = [3]$

$f([7,7,7]) = [7]$

$f$: ($\lambda x$ (singleton (third $x$)))

Figure 8: Concept c001 from Rule (2020).

Consider property "output_has_length_1" for task $c001$ (input-output examples shown in Figure 8 and properties shown in Table 1). Its estimated property value is AllTrue, that is $\Pi(p(f, S_i))$ = AllTrue. What does this imply about the property value for the latent program $f$? Either the latent program $f$ has property value AllTrue (hypothesis A) or it has property value Mixed and it is a coincidence that all example inputs result in True (hypothesis B). Both hypotheses are consistent with the observed data. We claim that as the number of examples observed increases (assuming A and B are still consistent with the data, i.e. all examples continue to have property value True), hypothesis A becomes exponentially more likely. Stated differently, B becomes exponentially more "suspicious" (Tenenbaum, 1999). Roughly, this is true because for programs in A, all inputs result in True for this property, whereas in B, only a subset of the valid inputs result in True for this property (by definition of Mixed). A more rigorous justification can be seen below.

The relative Bayesian posterior of A and B is given by,

$$\frac{\mathbb{P}(A \mid \Pi(p(f, S_i)) = \text{AllTrue})}{\mathbb{P}(B \mid \Pi(p(f, S_i)) = \text{AllTrue})} = \frac{\mathbb{P}(A) \cdot \mathbb{P}(\Pi(p(f, S_i)) = \text{AllTrue} \mid A)}{\mathbb{P}(B) \cdot \mathbb{P}(\Pi(p(f, S_i)) = \text{AllTrue} \mid B)}$$

We focus on the likelihood terms because only these scale with the number of examples. For A,

$$\begin{aligned}\mathbb{P}(\Pi(p(f, S_i)) = \text{AllTrue} \mid A) &= \sum_{f \in A} \mathbb{P}(f \mid A) \cdot \mathbb{P}(\Pi(p(f, S_i)) = \text{AllTrue} \mid f) \\ &= \sum_{f \in A} \mathbb{P}(f \mid A) \cdot 1^{|S_i|}\end{aligned}$$

For B,

$$\begin{aligned}\mathbb{P}(\Pi(p(f, S_i)) = \text{AllTrue} \mid B) &= \sum_{f \in B} \mathbb{P}(f \mid B) \cdot \mathbb{P}(\Pi(p(f, S_i)) = \text{AllTrue} \mid f) \\ &= \sum_{f \in B} \mathbb{P}(f \mid B) \cdot \left( \frac{\big|\{i \mid p(f, i) = \text{True}, i \in V(f)\}\big|}{\big|\{i \mid i \in V(f)\}\big|} \right)^{|S_i|} \\ &= \sum_{f \in B} \mathbb{P}(f \mid B) \cdot (1 - \epsilon)^{|S_i|} \quad \text{where } \epsilon > 0\end{aligned}$$

As the number of examples $|S_i|$ increases (assuming the corresponding property value continues to be True for all examples) the likelihood $\mathbb{P}(\Pi(p(f, S_i)) = \text{AllTrue} \mid B)$ exponentially decreases while the likelihood for A remains the same. Consequently, A quickly becomes more likely than B regardless of the prior probabilities $\mathbb{P}(A)$ and $\mathbb{P}(B)$. The above is fundamental to why we can accurately estimate property values even with a small number of examples.

**Future Work**

It would be interesting to explore how well the above Bayesian treatment of estimated property values could model human judgements about property values, especially using just a couple input-output examples. All the required quantities for the computational model can be estimated from the program prior. Additionally having a model that can quantify how good the estimated property values are could also be really useful for any synthesis algorithm that uses estimated properties (especially if the spec is only a couple of input-output examples).

## A.2 Handwritten Properties

All handwritten properties are either of type $(list(int), list(int)) \to bool$ or of type $list(int) \to bool$. The former are functions of both the input and output lists and the latter are functions of the output list alone. There is a total of 181 handwritten properties.

Each of the bullet points below corresponds to a single property:

- **output_els_in_input(input, output)**: Returns true if all output elements are present in the input.
- **input_els_in_output(input, output)**: Returns true if all input elements are present in the output.
- **output_same_length_as_input(input, output)**: Returns true if the output list has the same length as the input list.
- **output_shorter_than_input(input, output)**: Returns true if the output list length is strictly less than the the input list length.
- **output_list_longer_than_input(input, output)**: Returns true if the output list length is strictly greater than the input list length.

- **every_output_el_gt_every_input_same_idx_el(input, output)**: Returns true if every element in the output is strictly greater than the input element at the same index.

- **input_prefix_of_output(input, output)**: Returns true if the input is a string prefix of the output.

- **input_suffix_of_output(input, output)**: Returns true if the input is a string suffix of the output.

Each of the bullet points below corresponds to 10 properties for $k \in [0, 9]$:

- **output_contains_k(output)**: Returns true if the output contains the number k.

- **all_output_els_lt_k(output)**: Returns true if all the output elements are strictly less than k.

- **all_output_els_mod_k_equals_0(output)**: Returns true if all the output elements are multiples of k.

Each of the bullet points below corresponds to 11 properties for $n \in [0, 10]$:

- **output_list_length_n(output)**: Returns true if the output is length n.

Each of the bullet points below corresponds to 11 properties for $i \in [0, 10]$:

- **output_contains_input_idx_i(input, output)**: Returns true if the output contains the input element at index i (and it exists).

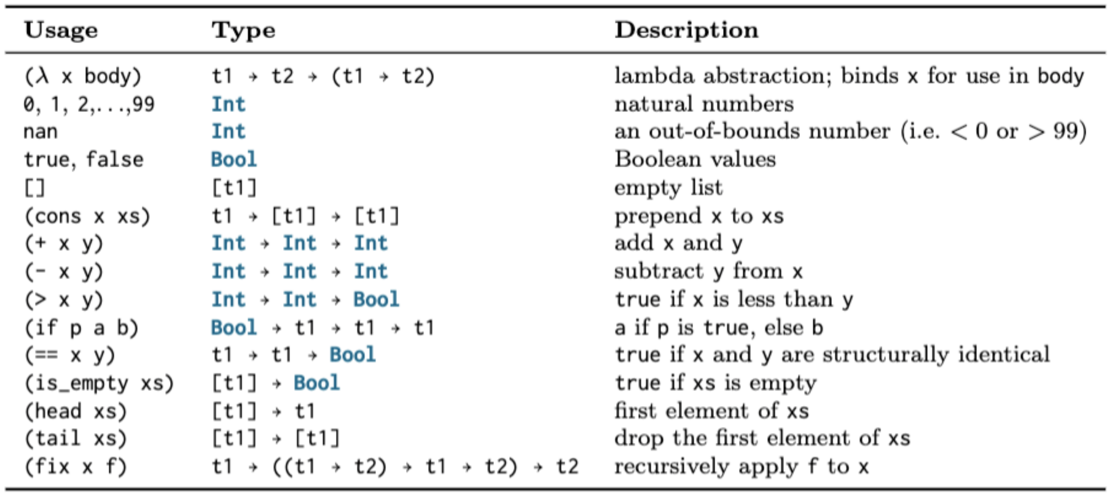
There are also 121 properties one for each $\{(i, j) \mid i \in [0, 10], j \in [0, 10]\}$ of the form:

- **output_idx_i_equals_input_idx_j**: Returns true if the output element and index i and the input element at index j exist and are equal.

We choose the values for $k, n, i, j$ based on the range of integers and the input and output list lengths in the Rule (2020) list function dataset (in our experiments we only use tasks with integers from 0 to 9). Datasets with longer lists, and larger integer ranges would result in many more properties. This is a limitation of the current formulation of properties of type $(list(int), list(int)) \rightarrow bool$.

## A.3 Josh Rule Basic DSL

The description of this DSL is taken directly from Table 6.1 of Rule (2020). For



| Usage | Type | Description |
|---|---|---|
| (λ x body) | t1 → t2 → (t1 → t2) | lambda abstraction; binds x for use in body |
| 0, 1, 2,. . .,99 | Int | natural numbers |
| nan | Int | an out-of-bounds number (i.e. $< 0$ or $> 99$) |
| true, false | Bool | Boolean values |
| [] | [t1] | empty list |
| (cons x xs) | t1 → [t1] → [t1] | prepend x to xs |
| (+ x y) | Int → Int → Int | add x and y |
| (- x y) | Int → Int → Int | subtract y from x |
| (> x y) | Int → Int → Bool | true if x is less than y |
| (if p a b) | Bool → t1 → t1 → t1 | a if p is true, else b |
| (== x y) | t1 → t1 → Bool | true if x and y are structurally identical |
| (is_empty xs) | [t1] → Bool | true if xs is empty |
| (head xs) | [t1] → t1 | first element of xs |
| (tail xs) | [t1] → [t1] | drop the first element of xs |
| (fix x f) | t1 → ((t1 → t2) → t1 → t2) → t2 | recursively apply f to x |

the experiments described in 6.2 and 7.2 we only use integers in the range 0 to 10 inclusive.

## A.4 Josh Rule Rich DSL

The description of this DSL is taken directly from Table 5.2 of Rule (2020).

| Usage | Type | Description |
|---|---|---|
| (λ x body) | t1 → t2 → (t1 → t2) | lambda abstraction; binds x for use in body |
| 0, 1, 2,...,99 | Int | natural numbers |
| true, false | Bool | Boolean values |
| [] | [t1] | empty list |
| (+ x y) | Int → Int → Int | add x and y |
| (- x y) | Int → Int → Int | subtract y from x |
| (* x y) | Int → Int → Int | multiply x and y |
| (/ x y) | Int → Int → Int | quotient of x divided by y |
| (% x y) | Int → Int → Int | remainder of x by y |
| (< x y) | Int → Int → Bool | true if x is greater than y |
| (> x y) | Int → Int → Bool | true if x is less than y |
| (is_even x) | Int → Bool | true if x is even |
| (is_odd x) | Int → Bool | true if x is odd |
| (and x y) | Bool → Bool → Bool | Boolean conjunction of x and y |
| (or x y) | Bool → Bool → Bool | Boolean disjunction of x and y |
| (not x) | Bool → Bool | Boolean negation of x |
| (if p a b) | Bool → t1 → t1 → t1 | a if p is true, else b |
| (== x y) | t1 → t1 → Bool | true if x and y are structurally identical |
| (singleton x) | t1 → [t1] | list with a single element, x |
| (repeat x n) | t1 → Int → [t1] | list repeating x n times |
| (range i j n) | Int → Int → Int → [Int] | list of numbers from i to j, inclusive, counting by n |
| (cons x xs) | t1 → [t1] → [t1] | prepend x to xs |
| (append xs x) | [t1] → t1 → [t1] | append x to xs |
| (insert x i xs) | t1 → Int → [t1] → [t1] | insert x at index i in xs |
| (concat xs ys) | [t1] → [t1] → [t1] | concatenate xs and ys |
| (splice ys i xs) | [t1] → Int → [t1] → [t1] | insert ys into xs, beginning at index i |
| (first xs) | [t1] → t1 | first element of xs |
| (second xs) | [t1] → t1 | second element of xs |
| (third xs) | [t1] → t1 | third element of xs |
| (last xs) | [t1] → t1 | last element of xs |
| (nth i xs) | Int → [t1] → t1 | element i of xs |
| (replace i x xs) | Int → t1 → [t1] → [t1] | replace element at index i in xs with x |
| (swap i j xs) | Int → Int → [t1] → [t1] | swap elements at indices i and j in xs |
| (cut_idx i xs) | Int → [t1] → [t1] | remove element at index i from xs |
| (cut_val x xs) | t1 → [t1] → [t1] | remove first occurrence of x from xs |
| (cut_vals x xs) | t1 → [t1] → [t1] | remove all occurrences of x from xs |
| (drop n xs) | Int → [t1] → [t1] | remove first n elements from xs |
| (droplast n xs) | Int → [t1] → [t1] | remove last n elements from xs |
| (cut_slice i j xs) | Int → Int → [t1] → [t1] | remove elements at indices i to j, inclusive from xs |
| (take n xs) | Int → [t1] → [t1] | first n elements of xs |
| (takelast n xs) | Int → [t1] → [t1] | last n elements of xs |
| (slice i j xs) | Int → Int → [t1] → [t1] | sublist of xs from indices i to j, inclusive |
| (fold f acc xs) | (t2 → t1 → t2) → t2 → [t1] → t2 | iteratively accumulate elements of xs into acc via f |
| (foldi f acc xs) | (Int → t2 → t1 → t2) → t2 → [t1] → t2 | like fold, but p is also given the element's index |
| (filter p xs) | (t1 → Bool) → [t1] → [t1] | keep only elements of xs for which p is true |
| (filteri p xs) | (Int → t1 → Bool) → [t1] → [t1] | like filter, but p is also given the element's index |
| (count x xs) | (t1 → Bool) → [t1] → Int | count occurrences of x in xs |
| (find p xs) | (t1 → Bool) → [t1] → [Int] | returns indices of xs for which p is true |
| (map f xs) | (t1 → t2) → [t1] → [t2] | apply f to each element of xs |
| (mapi f x) | (Int → t1 → t2) → [t1] → [t2] | like map, but f has access to each element's index |
| (group f xs) | (t1 → t2) → [t1] → [[t1]] | group elements, x, of xs based on the key, (f x) |
| (is_in xs x) | [t1] → t1 → Bool | true if x is in xs |
| (length xs) | [t1] → Int | length of xs |
| (max xs) | [t1] → Int | largest element in xs |
| (min xs) | [t1] → Int | smallest element in xs |
| (product xs) | [Int] → Int | product of elements in xs |
| (sum xs) | [Int] → Int | sum of elements of xs |
| (unique xs) | [t1] → [t1] | unique elements of xs |
| (sort f xs) | (t1 → Int) → [t1] → [t1] | sort elements, x, of xs by the output of (f x) |
| (reverse xs) | [t1] → [t1] | xs in reverse order |
| (flatten xs) | [[t1]] → [t1] | concatenates the list of lists, xs, into a list |
| (zip xs ys) | [t1] → [t1] → [[t1]] | join xs and ys into a list of two-element lists |

For the purpose of the experiments described in 6.2 and 7.2 we only use integers in the range 0 to 10 inclusive.

## A.5 Dreamcoder List Domain DSL

The primitives for this DSL can be found in the Dreamcoder repository[7].

## A.6 More general property formulation

A future direction to explore is to formulate properties to be of type $(list(int), list(int)) \rightarrow \tau$ where $\tau$ could be any of the DSL types. Then, instead of summarizing property values over specs with AllTrue, AllFalse and Mixed we would summarize with AllSame when $|p(f, S)| = 1$ and Mixed when $|p(f, S)| > 1$. With this formulation instead of requiring a property "output_list_length_n(output)" for every possible output list length, we would instead have a single property "output_length(output)". This would be sufficient to capture the "allSame" coincidences when all output lists of a given spec have the same length.

## A.7 Dreamcoder experiments

In this section we report some initial results from running Dreamcoder (Ellis et al., 2021) on the Rule (2020) list function dataset. Part of what makes the list function dataset challenging is that the basic DSL contains a small number of a very primitive functions (as can be seen in A.3 above). This is good reason to suspect that library learning could be beneficial as there is potential to invent richer primitives that shorten the program solutions. In fact the rich DSL in A.3 above is proof that such richer primitives exist. Additionally we run these these experiments as they can serve as a baseline for any future work that combines a property-based synthesis system with Dreamcoder.

We use the first 80 tasks from Rule (2020) that only contain integers from 0 to 10. All the Dreamcoder hyper-parameters are identical to the ones used for list domain experiments in the original paper and can be found in the original Dreamcoder

[7]https://github.com/ellisk42/ec/blob/master/dreamcoder/domains/list/listPrimitives.py#L352

repository[8]. We use the basic Rule (2020) DSL (rule basic dsl) and the list domain DSL used in the Dreamcoder experiments (dc list dsl).

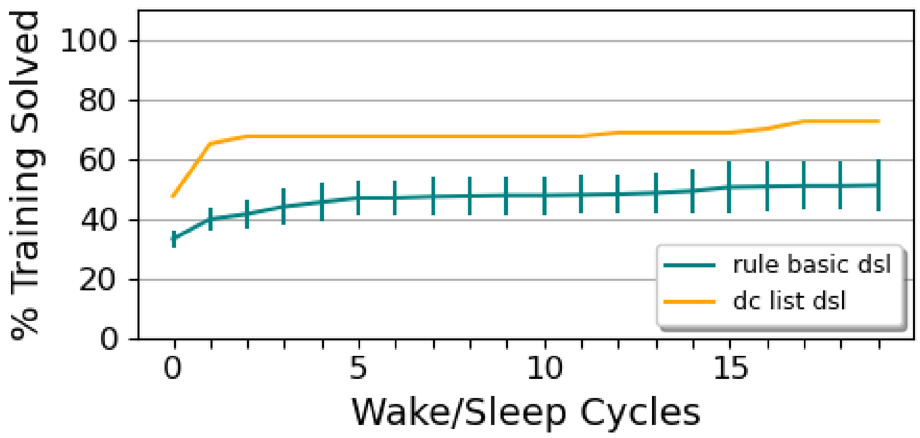


Figure 9: Percentage of tasks solved during library learning. We average 6 runs for the Rule (2020) basic DSL (rule basic dsl) and show the ±1 standard deviation error bars. Due to memory issues and time constraints we only run Dreamcoder once with the default Dreamcoder list domain DSL (dc list dsl).

In Figure 9 we see the percentage of the 80 training tasks solved through the course of library learning.

[8]https://github.com/ellisk42/ec/blob/master/official_experiments

## References

Samuel Acquaviva, Yewen Pu, Marta Kryven, Catherine Wong, Gabrielle E Ecanow, Maxwell Nye, Theodoros Sechopoulos, Michael Henry Tessler, and Joshua B Tenenbaum. Communicating natural programs to humans and machines. *arXiv preprint arXiv:2106.07824*, 2021.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

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.

Jerry A Fodor. *The language of thought*, volume 5. Harvard university press, 1975.

Noah D Goodman, Joshua B Tenenbaum, and Tobias Gerstenberg. Concepts in a probabilistic language of thought. Technical report, Center for Brains, Minds and Machines (CBMM), 2014.

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

Laurent Itti and Pierre Baldi. Bayesian surprise attracts human attention. *Vision research*, 49(10):1295–1306, 2009.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

Jinsuk Lim and Shin Yoo. Field report: Applying monte carlo tree search for program synthesis. In *International Symposium on Search Based Software Engineering*, pages 304–310. Springer, 2016.

Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. *arXiv preprint arXiv:2002.09030*, 2020.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*, 2020.

Steven Piantadosi. Fleet system. `https://github.com/piantado/Fleet`, 2020.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

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

Joshua Stewart Rule. *The child as hacker: building more human-like models of learning*. PhD thesis, Massachusetts Institute of Technology, 2020.

Joshua B Tenenbaum. Bayesian modeling of human concept learning. *Advances in neural information processing systems*, pages 59–68, 1999.

WD Vousden, Will M Farr, and Ilya Mandel. Dynamic temperature selection for parallel tempering in markov chain monte carlo simulations. *Monthly Notices of the Royal Astronomical Society*, 455(2):1919–1937, 2016.