CS 252r: Advanced Topics in Programming Languages

Eric K. Zhang
ekzhang@college.harvard.edu

Fall 2020

Abstract

These are notes for Harvard's *CS 252r*, a graduate seminar class on topics at the intersection of programming languages and artificial intelligence, as taught by Nada Amin[1] in Fall 2020. Possible topics include: differentiable programming, interpretable AI, neuro-symbolic systems, reinforcement learning, probabilistic programming, and synthesis.

Course description: We will explore programming languages for artificial intelligence. Programming languages drive the way we communicate with computers, including how we make them intelligent and reasonable. In this advanced topic course, we will look at artificial intelligence broadly construed from the point of view of programming languages. We gain clarity of semantics, algorithms and purpose.

Contents

[1] With teaching fellow: Mark Goldstein

1 September 3rd, 2020

We start with an overview of the course. This is a graduate seminar course with roughly 22 students, covering papers from a variety of topics. Students choose which papers they want to present, and grading is based on participation and a final project.

1.1 Breakout Discussion: Learning DSLs

Prof. Amin gives us the following open-ended discussion problem:

Example 1.1. Suppose that you have your favorite DSL (logic programming, string manipulation, regular expressions, etc.), and you want to devise a machine learning algorithm that learns a program from this language. How would you tackle this?

Our thoughts: Let's consider the examples of stack programming languages and regular expressions. We don't really know how to approach this problem since the space of programming languages isn't continuous or differentiable. A couple of thoughts:

- Initially, we can do a brute force search for all of the programs in a given length. With something like regular expressions, we might be able to search through examples of length less than 5, but we're stuck if we end up trying to look for something with length 20-ish.
- We can also guide our search by first creating examples from a generative model, rather than just brute force search. We can also have some sort of beam search approach.

These are pretty basic thoughts. Let's hear from Prof. Amin about recent approaches to program synthesis from various papers.

Note. Follow-up question: What if two different programs explain the set of examples?

Note. Follow-up question: How is learning a program different from encoding all the inputs and outputs numerically and applying generic machine learning?

1.2 Program Synthesis

There are four main approaches that we will be covering, and we will give some examples of research representatives in each approach.

1.2.1 Type-Driven Specification [OZ15]

Let's say that you have a functional programming language. One exercise you might be asked in a functional programming class is to write a `stutter` function, which simply takes a cons-list and doubles each element. For example:

```
type nat = O | S of nat
type list = Nil
          | Cons of nat * list
let stutter : list -> list |>
  { [] => [] | [0] => [0;0]
  | [1;0] => [1;1;0;0] } = ?
```

A "correct" program synthesized from this specification might look like:

```
let stutter : list -> list =
let rec f1 (l1:list) : list =
  match l1 with
  | Nil -> l1
  | Cons(n1, l2) ->
    Cons(n1, Cons(n1, f1 l2))
    in f1
```

This is a fairly simple example, but the key idea is that we are driving the synthesis of the program by *types* of variables. In this case, a list has two members: the first element and the rest of the elements, which can guide our search through the variable space. That is the intuition, but we can also formalize the four steps of this approach.

1. **Type Refinement:** From each type, we know input is type list and output is type list. Refine each example into a "world".
2. **Guessing:** Make a series of "guesses" of each leaf node of the tree, each within its own world with input and output variables. We also ensure that the output produces a valid constructor.
3. **Match Refinement:** Case analysis on an algebraic datatype, where you can split the examples accordingly.
4. **Recursive Functions:** Use examples as an approximation.

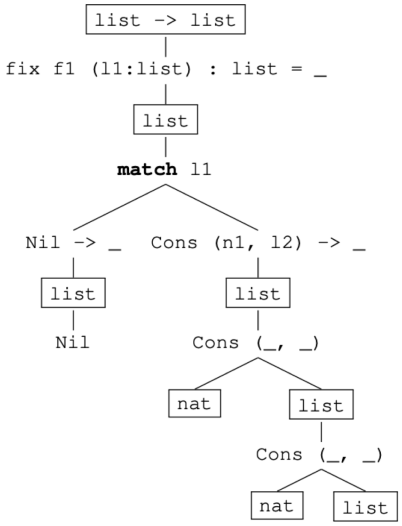


Figure 1: Example search tree from type-driven specification.

A couple limitations of this approach are that the examples must be *trace-complete*, i.e., they need to have a complete recursive call stack provided in the example inputs and outputs. This is an issue if you have a black box program, but you might still be able to get examples from guessing structural relationships. For example, the list of examples below is not trace-complete, but it would be if you removed the comment.

```
let list_stutter : list -> list |>
  { [] => []
  | [0] => [0;0]
  | [1;0] => [1;1;0;0]
  (*| [1;1;0] => [1;1;1;1;0;0]*)
  | [1;1;1;0] => [1;1;1;1;1;1;0;0]
  } = ?
```

This approach has worked fairly well for "introductory functional programming examples," which was pretty impressive when it came out. Another example where this succeeds is in something like an *arithmetic interpreter*. It doesn't perform well for more complex programs, however.

1.2.2 Logic-Based Approaches

Suppose that we have a set of observations represented as relations between data points, in an extensional database. Then we can write logic-based rules consistent with the rest of the trace of the input, which represent a basic program.

The object language here is *Datalog*$^{\supset}$, and the meta language is *Answer Set Programming (ASP)*. Our goal is to find the lowest cost cost(θ) for a given sequence, which is the total number of ground atoms in the initial set, plus the total number of unground atoms in the rules.

Example 1.2 (miniKanren, ICFP 2017)**.** This is a logic synthesis system in Racket, which can generate programs that produce a specific output. See http://minikanren.org/. Also, a demo solving functional programming problems with constraint logic programming is given here.

1.2.3 Neural-Guided Search [KMP+18]

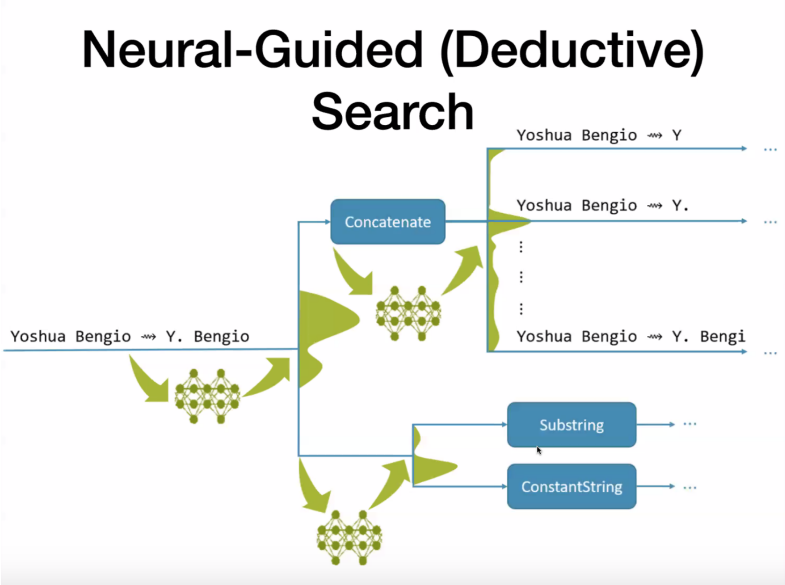


Figure 2: Neural Guided (Deductive) Search from Microsoft Research

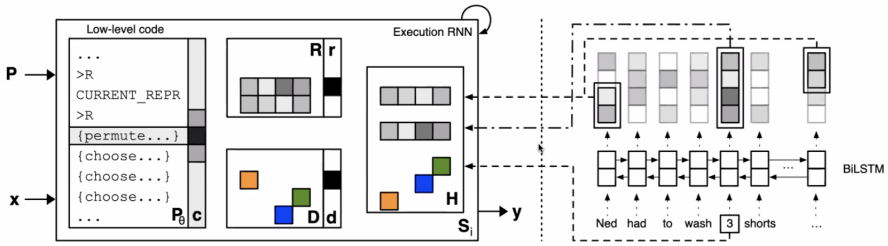
This approach is from Microsoft Research. At each step in your program synthesis algorithm, you use a neural network oracle to estimate the *most likely* next step. Given a grammar of possible operations, you can construct a tree of operations in each case.

The only issue here is: what policy do you use if the neural network guides you down the wrong path? It seems like this paper doesn't recover very well from missteps, but you could imagine a kind of beam search to try to do this better. Empirically, the paper reports about 10x speedups over previous methods though, which is not bad!

This is used in real life, see Microsoft Excel's *flash fill* tool that very responsively and automatically reformats data using string manipulation! The framework is quite nice to use yourself, and it allows you to create your own DSLs.

1.2.4 Differentiable Programming [BRNR17]

You can create a *differentiable interpreter* for a stack-based programming language. Your feature vector is represented by the *program counter*, which is fed into a recurrent neural network (*BiLSTM*) in each of several steps. This is allowed to modify the memory.




```
\ first copy data from H: vectors to R and numbers to D
{ observe R0 R-1 R-2 R-3 -> permute D0 D-1 D-2 }
{ observe R0 R-1 R-2 R-3 -> choose + - * / }
{ observe R0 R-1 R-2 R-3 -> choose SWAP NOP }
{ observe R0 R-1 R-2 R-3 -> choose + - * / }
\ lastly, empty out the return stack
```

Figure 3: Differentiable interpreter using BiLSTM

An interesting experiment from the paper is to look at grade school algebra word problems, and feed them into the differentiable Forth machine. Then, you can learn a differentiable programming language, evaluated by this RNN interpreter, that can understand and solve algebra word problems!

1.3 Breakout Prompts: Applying PL to Improve ML

Example 1.3 ([TBM+19])**.** Suppose you and a doctor want to predict patient blood glucose levels over time. You have lots of inpatient trajectories saved in the electronic health record and decide to use machine learning. Can you think of some constraints the doctor might know? How could we include this knowledge in our algorithm? Does your proposed approach guarantee, or only encourage, that the model will meet the constraints?

Our ideas (Eric Atkinson, Jiasi Shen):

- Add additional supervision, various loss functions for expected behavior.

- Lots of data augmentation, or reweighting of certain "textbook" examples.
- Locally model the trajectory as a sinusoid or other known function.

Example 1.4 ([SAB+19])**.** Suppose you have two syntactically-distinct programs that use randomness to supposedly approximate the same intractable quantity (e.g. some integral). How can you define correctness? Is one of them "better"? In what sense could these programs "behave in the same way" or be semantically "the same"?

Once again, brainstorming some ideas:

- Eric mentions he's dealt with MCMC algorithms before, where multiple complicated random models *should* converge to the same distribution, but there's no *a priori* way to tell. For example, Gibbs sampling, Metropolis-Hastings, or Hamiltonian Monte Carlo.
- Test on a bunch of small examples, or a domain, and make sure the means and variances empirically match?
- MCMC algorithms have exponentially large spaces, so it's hard to verify initially. But maybe you could get away with random projections to a d-dimensional real space, then comparing the distributions of the algorithms.

Final notes for next time: auditors do not need to present a paper, but we have the freedom to discuss whichever papers we find interesting.

2 September 8th, 2020

Today, Yizhou Yang (Assistant Professor, Waterloo) will present a guest lecture on probabilistic programming, and Mark Goldstein (TF) will present a lecture on Bayesian inference. There will also be breakout sessions and discussions.

2.1 Probabilistic Programming [vdMPYW18]

Probabilistic programming is about *drawing random values from distributions*, and *computing information* about those values. The second part of this definition is very important, as without it, we could claim to be doing probabilistic programming by just using a random number generator like `import random` in Python!

Key concepts in randomness include:

- **Conditioning**: specifying "good" executions that match observations, using Bayes' Rule:

$$\Pr(X \mid Y) = \frac{\Pr(X, Y)}{\Pr(Y)} = \frac{\Pr(Y \mid X)\Pr(X)}{\Pr(Y)}.$$

  In other words,

$$\text{posterior} = \frac{\text{likelihood} \cdot \text{prior}}{\text{normalizing factor}}.$$

  A lot of the work that probabilistic programming languages do for you is to help you compute things like the *normalizing factor*, which are annoying or difficult to get by hand.

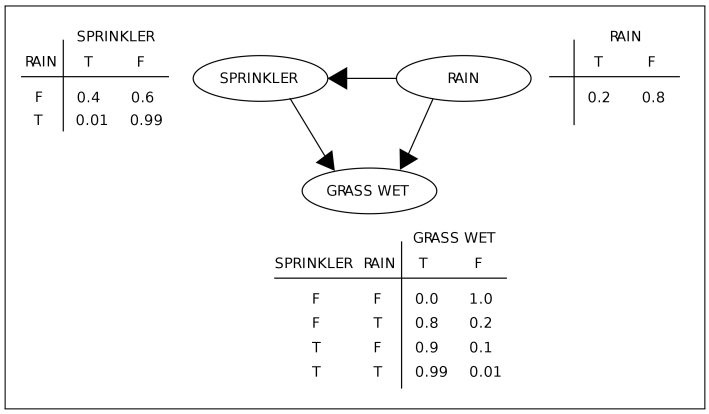


Figure 4: Example Bayesian network.

Given a *Bayesian network* made of several tables such as the one shown here, one can answer both simple and complex queries, such as the following:

- $\Pr(W \mid S, R) = 0.99$,
- $\Pr(W \mid R) = \Pr(W, S \mid R) + \Pr(W, \overline{S} \mid R) = 0.8019$,
- $\Pr(R \mid W) = \frac{\Pr(R)\Pr(W|R)}{\Pr(W)} = 0.3577$.

However, these kinds of manual calculations quickly get error-prone and tedious. Instead, we can automate this process by using a probabilistic programming language like *WebPPL*, which is a JavaScript web-based programming environment.

```
var geometric = function() {
  return flip(.5) ? 0 : geometric() + 1;
}

var conditionedGeometric = function() {
  var x = geometric();
  factor(x > 2 ? 0 : -Infinity);
  return x;
}

var dist = Infer(
  {method: 'enumerate', maxExecutions: 10},
  conditionedGeometric);

viz.auto(dist);
```

In traditional statistical modeling and inference, we have three steps: developing inference algorithms, writing down a model, and computing outputs. In contrast, probabilistic programming lets you evaluate models through an *interpreter*, which can furthermore be embedded within a host language like JavaScript. In this way, you get all the features of the host language for free, which gives you more flexibility.

Example 2.1. Given a prior distribution on the parameters, you can naturally do Bayesian linear regression with probabilistic programming.

```
var xs = [0, 1, 2, 3];
var ys = [0, 1, 4, 6];

var model = function() {
  var m = gaussian(0, 2);
  var b = gaussian(0, 2);
  var sigma = gamma(1, 1);
  var f = function(x) {
    return m * x + b;
  };
  map2(
    function(x, y) {
      factor(Gaussian({mu: f(x), sigma: sigma}).score(y));
    },
    xs,
    ys);
  return f(4);
}

viz.auto(Infer({method: 'MCMC', samples: 10000}, model));
```

Example 2.2. Suppose that you have a virtual physics environment, with a falling ball, a goal bucket, and three arbitrarily positioned "bumpers". Using a simple physics simulation, we can

condition the distribution of the bumper locations on the event that the *ball falls into the bucket*, which lets us sample from this distribution of bumpers!

These examples show how PPLs (probabilistic programming languages) are at the intersection of programming languages, statistics, machine learning, and artificial intelligence. Some are standalone, but most are embedded in larger general-purpose programming environments for flexibility. A non-exhaustive list includes: Pyro in Python/PyTorch [BCJ+19], Stan in R, Infer.NET in C#, Anglican in Clojure, WebPPL in JavaScript, and ProbLog.

However, we generally have a tradeoff in the design of PPLs between *expressive power* and *inference efficiency*. Some PPLs like Stan and Infer.NET do not give you full access to loops and black-box functions from the enclosing language, to enable efficient inference (some sort of closed-form, reversible analytic solution). However, others like WebPPL let you use black-box functions and have various sampling strategies instead. See `http://dippl.org` for more on design.

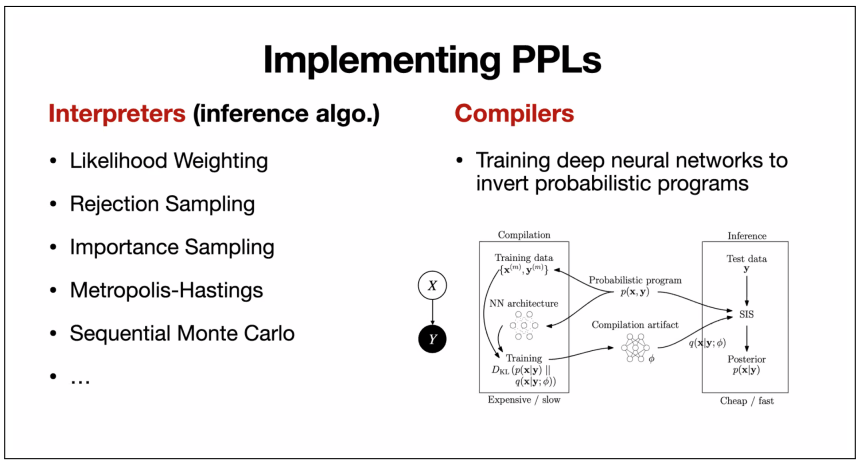


Figure 5: Different modes of evaluating PPLs: inference and compilation.

Key takeaway: PPLs are a programming language abstraction that allow you to decouple probabilistic *modeling* from *inference* in a natural, automated way.

2.2 Bayesian Inference Algorithms

Let's talk more about what a probabilistic model really is. Our setup is that we have some observed data $x \in X$, and we want to infer some unobserved structure $z \in Z$. There are two ways to think about the problem of modeling:

- Specify some way to measure the *compatibility* $p(x, z)$ between two variables.
- Specify the data generating process, where you have some sampled $z \sim p(z)$, and then you sample $x \sim p(x \mid z)$. You didn't observe this process and so only have x.

These two problem statements are equivalent mathematically since $p(x, z) = p(x \mid z)p(z)$.

Example 2.3. Some problems with unobserved z and observed x:

- Given a robot's noisy position measurements x from sensors, predict its true position z.
- Given a person's test scores, predict their IQ or numerical rating.

The key problem is that if we want to sample values from the conditional distribution $p(z \mid x)$, we can model this distribution using Bayes' Rule:

$$p(z \mid x) = \frac{p(x, z)}{p(x)} = \frac{p(z)p(x \mid z)}{\int_Z p(x \mid z') \, \mathrm{d}z'}.$$

However, the bottom integral (*normalizing factor*) can be difficult to compute when our state space is large, such as when there are many dimensions. This motivates developing the celebrated Metropolis-Hastings algorithm[2] for Markov chain Monte Carlo (MCMC), which implicitly generates a Markov chain over X whose transition probabilities are easy to compute, and whose stationary distribution is equal to $p(z \mid x)$.

[2] Apparently, Natesh Pillai really likes this algorithm.

3 September 10th, 2020

Today, Joey Velez-Ginorio will present a lecture on "Differentiable Programs with Neural Libraries" [GBKT17]. Helpful prior literature includes "TerpreT" [GBS+16].

3.1 Breakout Discussion: Perceptual Versus Symbolic

Exercise 3.1. Which problems are perceptual, versus symbolic? Are there problems that are all purely perceptual, or purely symbolic, or what are the clear lines that divide these two parts?

Discussion between Chris Hamblin, Anastasiya K., Shashank Srikant, Hope Kean, and me:

- Some problems are purely symbolic (computers get symbolic inputs, make symbolic outputs).
- Addition example: perceptual aspects of assigning a numerical digit to each image, then symbolic aspect of performing addition.
- Classical ML / "expert systems" = try to force perceptual problems into the symbolic domain, gaining explainability at the expense of accuracy and simplicity.
- Generally symbolic = explainable, and end-to-end perceptual is not really explainable.

Why do we care about breaking up problems between perceptual and symbolic domains? It's because sometimes, getting training data is easier in the *joint* sense than the *separate* sense.

For example, in image captioning, there's a perceptual component, as well as a more logical component of producing descriptions. We can only get training data for the end-to-end system of producing captions, however.

3.2 Differentiable Programming with Neural Libraries [GBKT17]

You can write programs in a language called *TerpreT* (similar syntax to Python), which combines differentiable operations with neural network layers. This allows you to train a joint perceptual-symbolic system using backpropagation.

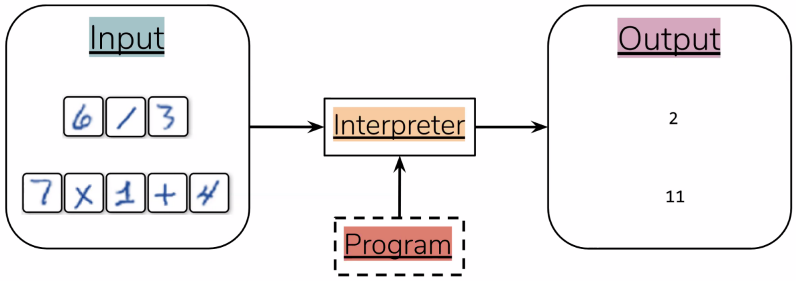


Figure 6: Calculator problem solved by Neural TerpreT.

Important details about this language is that although it supports operations like `for` loops and conditionals, being differentiable is very difficult. For example, how do you differentiate integers? To handle this, TerpreT uses the following differentiable reparameterization trick:

- At the beginning of the program, initialize a "runtime" with constant `INT_SIZE`, such as 10.

- Replace every integer variable with a fuzzy vector of probabilities with `INT_SIZE` elements, one probability for each discrete value.
- Compile each conditional into a linear operator.

The language is kind of "stupid simple" in the sense that all you can do is conditionals, loops, arithmetic, and matrix multiplication (i.e., fully-connected neural networks).

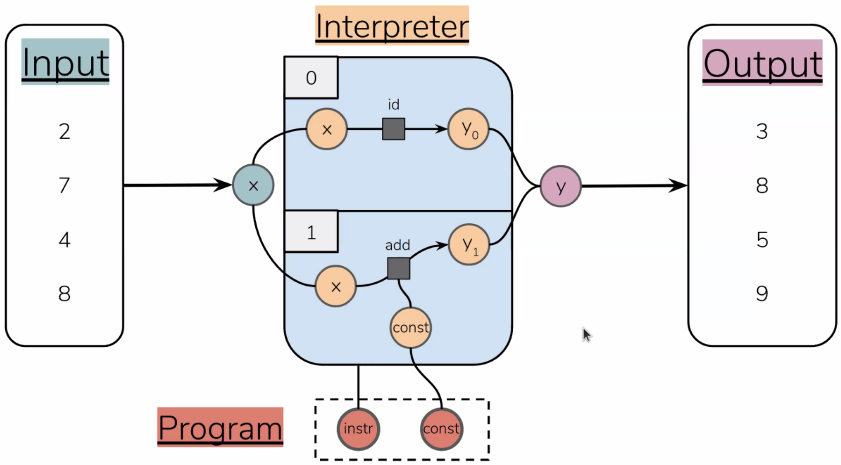


Figure 7: Reparameterization of discrete values makes conditionals differentiable.

See the Fig. 7 for an example of how conditionals can be differentiated by representing each discrete variable as a *probability distribution* over its possible values. This also happens to make the output differentiable, and we can train the neural interpreter using gradient descent on cross entropy loss. Mathematical details can be found in [GBS+16].

Question: Why constrain yourself to one of these two methods in this case, toggled by a conditional? Can you do synthesis on a prior distribution over all programs constructible from a DAG of instructions? (Answer: maybe, but it seems hard)

The point of *Neural TerpreT*, compared to generic TerpreT, is that you can add operators that correspond to neural networks. You will see that this is a common pattern among many differentiable programming languages: someone initially creates the differentiable operator set, and a follow-up paper adds neural networks to it.

The previous example was just a very basic conceptual point, but more promising directions for this system are actually based on differentiably learning *assembly* instructions.[3] Fig. 8 shows how a Neural TerpreT program can follow directions in a grid of sign images, backpropagating over the entire path, rather than learning individual sign directions.

> Source code representation enforces an inductive bias that favors learning solutions that exhibit strong generalization. For example, once a suitable control flow structures (e.g., a for loop) for a list manipulation problem was learned on short examples, it trivially generalizes to lists of arbitrary length.

Question: Can you actually learn programs of this form, as they get more complicated? I'm not sure if delegating several operations to the neural network allows backpropagation to work on the coarse-grained training approach we have here. Does this scale / do you have enough supervision?

[3]Note a limitation: you need to know how many instructions there are, ahead of time.

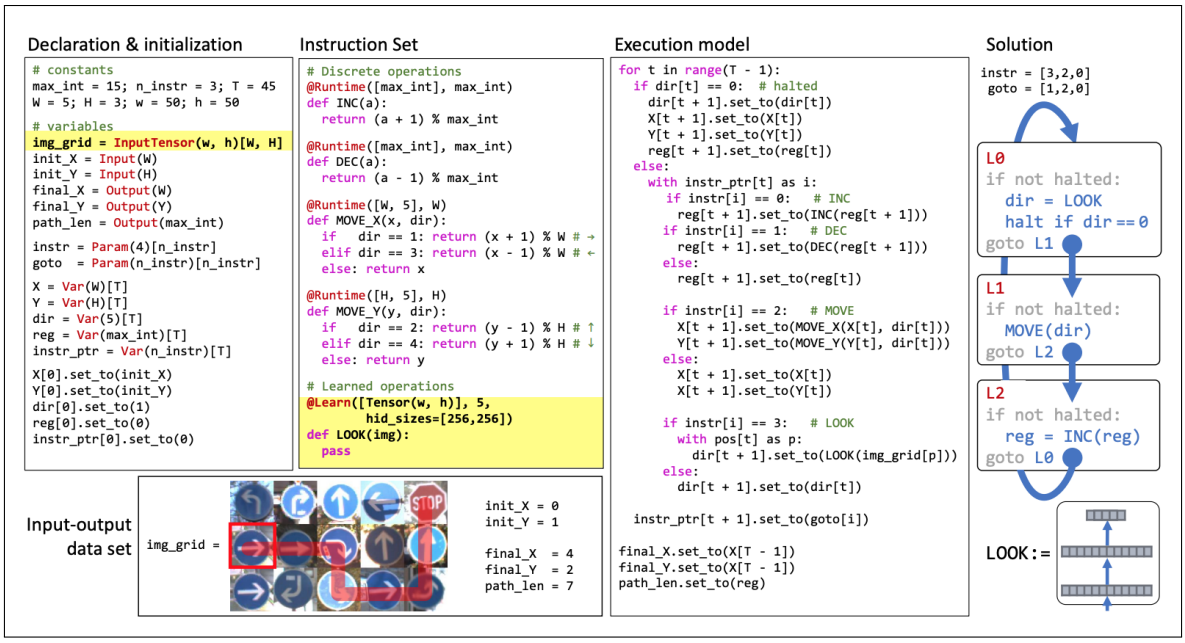


Figure 8: Differentiable assembly-like operations in TerpreT for a perceptual graph problem.

An idea I was toying with is adding additional supervision to the programs you generate, such as *differentiable static analysis*. The idea is to make it easier for the synthesizer to generate reasonable programs (e.g., not having infinite loops or useless variables).

Finally, a major point of the neural differentiable programming scheme is that it facilitates *transfer learning*, where you can solve related problems that differ symbolically while using the same perceptual parts. They do some experiments that show *catastrophic forgetting* does not occur when attempting to transfer knowledge to new tasks; accuracy generally increases.

Summary: Neural differentiable programming gets great performance in transfer learning and generalization on toy problems, compared to baseline models (proof of principle).

4 September 15th, 2020

Today, Priyan Vaithilingam will present a talk on "Neuro-Symbolic Program Synthesis" [PMS+16].

4.1 Basic Program Synthesis

The problem of *program synthesis* is roughly the following: given a specification of a problem, either English-language or input/output examples (as in this paper), automatically produce a candidate program using an algorithm. There are three axes to this problem:

- Behavioral Constraints: formal specification, or input-output examples;
- Structural Constraints: DSL, or grammar of the program to learn;
- Search Strategy: learning algorithm used.

$$
\begin{array}{rcl}
\textbf{String } e & := & \text{Concat}(f_1, \cdots, f_n) \\
\textbf{Substring } f & := & \text{ConstStr}(s) \\
 & | & \text{SubStr}(v, p_l, p_r) \\
\textbf{Position } p & := & (r, k, \text{Dir}) \\
 & | & \text{ConstPos}(k) \\
\textbf{Direction Dir} & := & \text{Start} \mid \text{End} \\
\textbf{Regex } r & := & s \mid T_1 \cdots \mid T_n
\end{array}
$$

Figure 9: Grammar of a simple string-processing language used in the paper.

Exercise 4.1 (Breakout discussion)**.** How might you use a basic supervised machine learning algorithm to tackle this problem?

Thoughts on this (Alex Gu, Anthony Yim, Mark Goldstein, Yuan Cao):

- Grammar-directed search for programs eliminates compilation errors.
- Generating programs token-by-token with some goal-directed supervision.
- Neural network to complete partial programs, incorporated into loss.
- We prefer continuous (differentiable) loss functions because we can use backpropagation and gradient descent, but a lot of traditional interpretation is discrete.

All of these are relatively basic ideas, but thinking about this problem is helpful to bring into context the discussions from Section 1.2.

4.2 Neuro-Symbolic Program Synthesis [PMS+16]

The key idea in this paper is that many program synthesis approaches are complex, expensive, and require retraining for each problem you want to solve. In contrast, this paper proposes a synthesis algorithm that is trained once and generates programs at *inference* time.

Definition 4.1 (Recursive-reverse-recursive neural network)**.** The novel architecture in this paper is the *R3NN*, a neural network that takes a partially-developed program syntax tree as input, then expands non-terminals incrementally into a full derivation based on a context-free grammar.

Each leaf symbol and production rule is given an M-dimensional vector representation, while each production rule is assigned a deep neural network $f_r : \mathbb{R}^{Q \cdot M} \to \mathbb{R}^M$, where Q is the arity of the rule. We also assign each production rule a *reverse* neural network $g_r : \mathbb{R}^M \to \mathbb{R}^{Q \cdot M}$, which is used to predict the representations of the generated symbols on the RHS of the production rule.

At each step, the expansion rule probabilities are then calculated based on both these representations, as well as the input-output string pairs, which are passed into a bidirectional LSTM. The two steps are roughly:

- (Recursive) Develop a global structure representation for the entire tree using f_r.
- (Reverse-Recursive) Add the global structure into the representation of each node, then pass this composite representation top-down through the tree by g_r.

A diagram of the recursive step is shown below. Here, each terminal symbol s has already been assigned a representation $\phi(s)$, and the neural networks f_r with weights W_r and activation function σ are used to recursively calculate ϕ for the non-terminal subtrees.

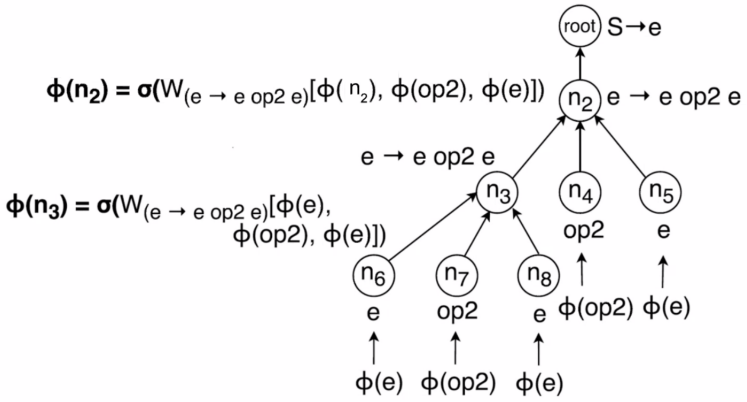


Figure 10: Recursive step of the R3NN model.

The main idea of this recursive step is that you want some global representation of the program tree you've constructed so far. In the reverse-recursive step, each reverse neural network g_r with weights W_r is used to compute representations $\phi'(s)$ for each node s in the tree, by initializing $\phi'(root) = \phi(root)$ and propagating it down. For every node n with children $c_1, \ldots, c_Q$,

$$[\phi'(c_1), \phi'(c_2), \ldots, \phi'(c_Q)] = \sigma(W_r \phi'(n)).$$

Finally, the sampling probabilities of creating each expansion e with non-terminal leaf node ℓ and rule r is simply $z_e = \phi'(\ell) \cdot \omega(r)$, where $\omega(r)$ is the M-dimensional vector representation of rule r. We can then pass these through a softmax function and sample!

Exercise 4.2. Question for breakout rooms: how does training work for this model?

It seems like a reasonable training procedure would be to just take a bunch of example derivations for trees (right now we haven't covered input-output yet), and do maximum likelihood estimation (i.e., cross entropy / empirical risk minimization).

Note. This model cares about the order in which leaves are derived (we pick the order with softmax), so why can't we just always take the leftmost derivation instead? This would make this essentially a stack-based generative model, which seems like it would be simple, as the derivation order shouldn't matter.

So far, we've talked about the *marginal distribution* of the program trees without considering *conditioning* on input-output pairs. Let's now discuss how to estimate the conditional distribution. We pass each input-output pairs into a BiLSTM to get a continuous representation, and then there are three ways to proceed:

- Pre-conditioning: Add representations to each leaf node before the recursive step.
- Root conditioning: Add representations to the root node $\phi(root)$ between steps.
- Post-conditioning: Add representations to leaf nodes after the reverse-recursive step.

Empirically, they saw that pre-conditioning worked best. Their results section is quite complicated and not as good as *FlashFill* (length limitations of learned programs), but still fairly impressive (after all, published in ICLR 2017). Also, they showed by comparison that they perform better than a naive, text LSTM program synthesis model, which doesn't consider structure.

5 September 17th, 2020

Today, Chris Hamblin will present a lecture on "DeepBach: A Steerable Model for Bach Chorales Generation" [HPN17].

5.1 DeepBach [HPN17]

Bach chorales are an interesting medium for music modeling with AI, for three reasons: structure, simple yet rich harmonic language, and agnostic to the instruments / tone colors used. This paper aims to solve the pragmatic task of creating a chorale editor that generates chorales, fills in gaps, and allows for conditioning on melodies or other parts.

The core algorithm is extremely simple (for a deep learning algorithm). Roughly speaking, you just format a chorale into a simple string consisting of notes in sixteenth-note pulses, with associated metadata like fermatas (representing cadences in Bach). You then pass this string into a BiLSTM with one note (sixteenth-note time point) removed, and try to predict that column based off the notes before and after it. With this conditional distribution, you just pass it into "pseudo"-Gibbs sampling and win, i.e.:

- Select a random note V_i^t, meaning voice i at time t.
- Delete the old note and resample it according to the conditional distribution on the rest of the notes (i.e., the output of the BiLSTM at that position).
- Repeat until reaching the steady state distribution.

The paper makes some justifications for allowing the model to be *time-agnostic*, but we do train distinct parameter tensors θ_i for each voice i.

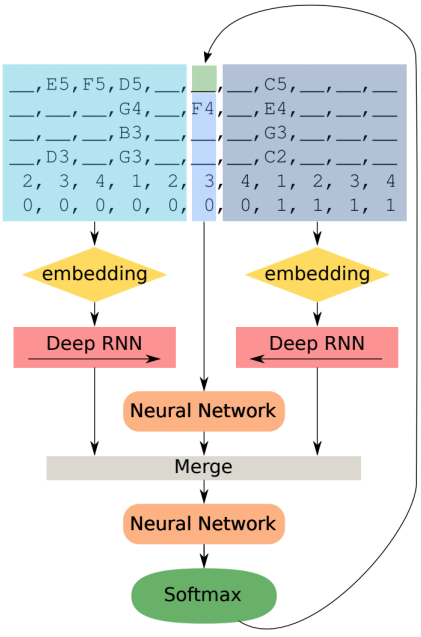


Figure 11: Graphical representation of DeepBach's neural network architecture.

5.2 Steerability of Gibbs Sampling

DeepBach is *steerable* in the sense that their sampler can choose to only resample notes from one or more voices, or notes in a specific region of the music. This allows them to create a neat MuseScore plugin that lets you select a region and only rewrite notes in that selection. Your initial values could either be sampled from the marginal distribution or supplied by the user.

Other clever tricks are, for example, only sampling notes that are not blank (underscores), which fixes the rhythm, or doing the opposite, which adds non-chord tones! There's a lot of fun sampling methods related to this idea. You could potentially consider this the most useful, or novel, contribution of the paper.

Note. The reason why they call this *pseudo*-Gibbs sampling is that no joint distribution actually exists for the conditional distributions produced by the model. It turns out that this doesn't converge on a universal joint distribution due to violating some probability conditions [AP89], but in practice, the algorithm works fine anyway.

More interesting, modern work at Google AI using transformers can be seen in [HVU+18].

6 September 22nd, 2020

Today, Andrew Kim will present the paper "Modeling Semantic Cognition as Dimensionality Reduction". The main field of this paper is computational cognitive science (modeling human cognition), which may have ties to program synthesis.

6.1 Modeling Semantic Cognition [KGK+08]

In the computational neuroscience setting, we are interested in relations between abstract objects with certain structures. This motivates the following definition:

Definition 6.1 (Theory, neuroscience version)**.** A *theory* is any structure that organizes a domain of knowledge according to underlying laws. Examples of theories include taxonomies, ontologies, or family trees.

This paper focuses on two main representations of relations: symbolic and connectionist. Whereas the symbolic representation is minimalist and perfectly reflects the data when combined with the rules, the connectionist approach prefers to generate all possible relationships, without explicitly representing those rules. Generally, the connectionist approach handles noisy data and exceptional cases better, but sometimes misses "obvious" connections in the data.

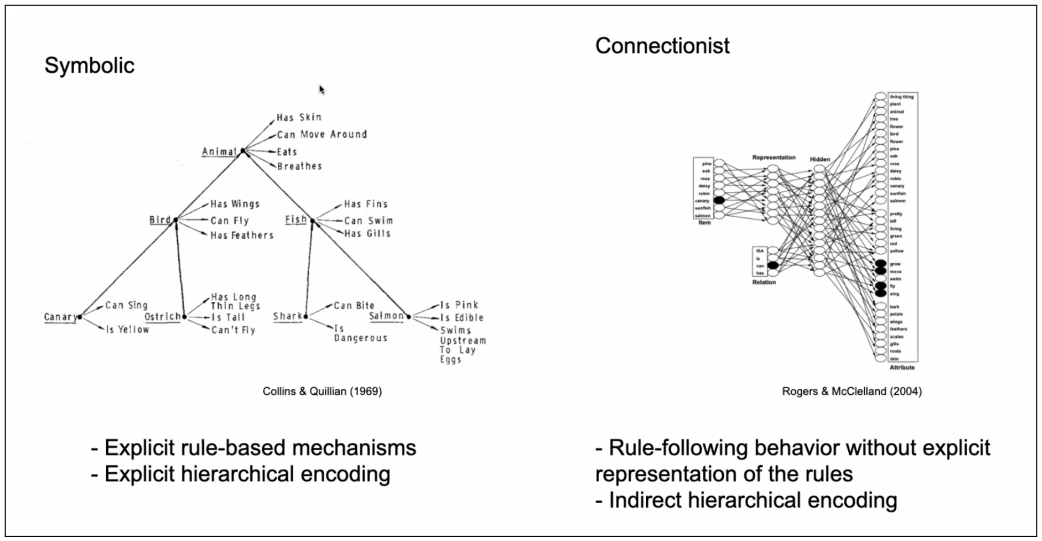


Figure 12: Comparison of connectionist and symbolic models.

The method starts from *core relations*, reminiscent of extensional database (EDB) relations in Datalog. For example, in the family tree example, you can start with three core relationships:

- `female(X)` - whether person `X` is female,
- `spouse(X, Y)` - whether `X` and `Y` are married,
- `child(X, Y)` - whether `Y` is a child of `X`.

From this point, you can use standard Horn clauses (with negation) to derive additional intensional relations like `father(X, Y)` and `daughter(X, Y)`, as well as to fill in more incomplete relationships in the input. For example,

```
spouse(X, Y) :- spouse(Y, X).
child(X, Y) :- child(Z, Y), spouse(X, Z).
father(X, Y) :- child(X, Y), !female(X).
daughter(X, Y) :- child(Y, X), female(X).
```

6.2 Generating Contexts via a Bayesian Approach

Now let's start talking about how one might *learn* a system such as this one. At a first level, we want to answer the following question. Given some observations, what is the most likely model? Luckily, we can generate a distribution of possible observations from a theory (by sampling the *context*, i.e., core relations, from Bernoulli(p) where $p \sim \text{Beta}(\alpha, \beta)$). If we push forward through the Horn clause interpreter, this yields a probability of generating each output.

It's left a little unclear how the model is actually discovered. Anyway, it probably has to do with some Bayesian sampling algorithm like MCMC. Once we discover the model, we have a couple of advantages over previous approaches (symbolic and connectionist):

- **Uncertain, graded inference:** You can pass in each input with a certain level of uncertainty ($p = 0.1$) and still compute outputs based on it. Also, if your data is slightly wrong, you can still obtain correct outputs with high probability.
- **Common-sense interfaces:** Conceptually simpler than connectionist models since the number of dimensions, or core relations, is reduced.

To clarify, the initial laws (or Horn clauses) are provided as an *input* to the learning algorithm, while our search space is actually just the *context*, i.e., the set of *core relations*. This is totally combinatorial, and we simply sample a ton of contexts until we find the one that's best. Inference given probabilistic core relations can be computed with MCMC.

6.3 Generating Theories Given a Context

In this second part of the paper, we analyze a complementary problem. Given a context (initial set of core relations) and observed relations, how can we learn a theory that represents the connections between these?

This paper doesn't seem to be terribly practical. It proposes a very limited search space of theories (basically, given 3 candidate laws, choose from all 2^3 subsets). We also sample over all combinations of core relations (unary, binary). Since the problem is very simple and supervised, we can do just do an exhaustive search on all combinations of laws. When we score the value of a law, we scale its likelihood by $2^{-\ell}$, where ℓ is the representation length.

Note. Joey notes that the goal of this paper was not to solve an *engineering task* like many of the other papers in this seminar. Since it was published in a purely cognitive science journal, the point was more to *capture human performance* and understand how humans reason, rather than to be efficient in the computational sense. However, as Anastasiya mentions, it's still a little bit weird that there's no comparisons made to previous models.

Perhaps this paper's main contribution is that people hadn't thought this way about cognitive science before, and so this was more of a proof-of-concept to introduce a new idea. For more work in this field, see some of Josh Tenenbaum's later publications, including [TKGG11]. His group essentially spent the next 10 years working on probabilistic and differentiable programming, including the *Church* language [GMR+12].

7 September 24th, 2020

Today, Prof. Amin gives a talk on "The Simple Essence of Automatic Differentiation" (AD), related to formalizing an automatic differentiation system in Haskell [Ell18]. Other relevant papers include [ZDW+19, AP20].

7.1 Category-Theoretic Automatic Differentiation [Ell18]

The crux of this paper is defining a functional programming system for computing the derivative of functions. Consider a function $f : a \to b$. Then, we define a typeclass $D\ a\ b = a \to (b \times (a \multimap b))$, so that we can define the functor $\mathcal{D}^+ : (a \to b) \to D\ a\ b$ as

$$\mathcal{D}^+ f\ a = (f\ a, \mathcal{D} f\ x).$$

This is indeed a functor, as we can write the chain rule as a composition of differentiation functors in the D category. In other words, where $\circ$ is efficiently a composition of morphisms,

$$\mathcal{D}^+(g \circ f)\ a = \textbf{let}\ \{(b, f') = \mathcal{D}^+ f\ a; (c, g') = \mathcal{D}^+ g\ b\}\ \textbf{in}\ (c, g' \circ f').$$

In particular, many functions can be implemented in this form. We can easily generalize to so-called *Cartesian categories* by writing

$$\mathcal{D}^+(f \times g)(a, b) = \textbf{let}\ \{(c, f') = \mathcal{D}^+ f\ a; (d, g') = \mathcal{D}^+ g\ b\}\ \textbf{in}\ ((c, d), f' \times g').$$

Therefore, we have introduced so-called *sequential composition* and *parallel composition*, which allows us to duplicate variables and chain functions together. Using these two operations, we can implement any function in this automatic differentiation framework by defining its instance as a Haskell typeclass, and we get full automatic differentiation for free![4]

7.2 Breakout Discussion: Compositionality in PL and ML

Discussion with Shashank, Priyan, and Hope:

- Composition is everywhere, no one implements a complex neural network from scratch by manually writing every layer.
- Many different definitions of the word *composition* in different fields. Even within programming languages, no one has a precise definition.
  - In denotational semantics, composition means that you can use the parts as a whole.
  - However, you still have a tradeoff for *extensibility*, where you can parameterize the parts.
- Many people use black-box models composed with others (e.g., BERT).
- Compositionality can be used as an inductive bias for the designing of deep systems. At the lowest level, compositionality exists at the layers implemented by a deep learning library.

Although this expression of automatic differentiation as a category is not necessarily more *computationally efficient*, Nada notes that it can be a nice generalization. Once you have a program expressed in the language of categories, you can generalize automatic differentiation to a whole class of different operations by "simply" changing the implementation of some typeclasses.

[4] See also [Ell17] for an earlier paper from the same author on automatically compiling a subset of Haskell programs to category theory.

7.3 Generalized Automatic Differentiation

Although we started from a simple formulation in terms of derivatives as linear maps $a \multimap b$, we can generalize to any linear function $\rightsquigarrow$ by parameterizing the category $D_{(\rightsquigarrow)}$ and re-implementing the *mul* operation. See the paper for more details on this; it involves defining a Cartesian cocategory over the *scale* and *add* operations.

We can also build arbitrary matrices with the $\triangle$ and $\triangledown$ operations, which horizontally and vertically juxtapose fragments of matrices. With this, we can eventually end up with automatic differentiation in the general matrix setting, as long as you have a definition of the $(\rightsquigarrow)$ arrow. Prof. Amin now spends a few minutes talking about really advanced category theory and functional programming stuff, but this unfortunately went over my head.

Definition 7.1 (Continuation passing style)**.** Recall the definition of a dual vector space. We can write a linear map $a \rightsquigarrow b$ as $(b \rightsquigarrow r) \to (a \rightsquigarrow r)$; this is the so-called *continuation passing style (CPS)* from functional programming, where you pass around a handle to future computation being done. This is similar to the Yoneda lemma in category theory.

Anyway, there's some fancy thing being done here to re-represent categories in the reverse direction, by associating $a \multimap b$ as $(b \multimap r) \to (a \multimap r)$, which then becomes $b \to a$.[5] This may have some performance benefits for the category compiler.

For example, in the continuation passing style (callbacks), the list *reverse* function can be implemented in $O(n)$ time instead of the naive $O(n^2)$ time. Here's an example in Scheme:

```
; Naive O(n^2) version
;   reverse: list -> list
(define (reverse as)
  (if (null? as) '()
      (append (reverse (cdr as)) (list (car as)))))

; Tail call O(n) version
;   reverse1: list -> list -> list
(define (reverse1 as res)
  (if (null? as) res
      (reverse1 (cdr as) (cons (car as) res))))

; Continuation passing style (CPS)
;   k: list -> r
;   reverse2: list -> (list -> r) -> r
(define (reverse2 as k)
  (if (null? as) '()
      (reverse2 (cdr as) (lambda (res) (k (append res (list car as)))))))
```

Continuation passing style is a good target language for compilers, for the reason that it clarifies the order between operations. I believe it's actually used a lot in industry, such as in Node.js's callback-based API, which is analogous to this. This is future food for thought!

[5] This is somewhat weird notation. I'm pretty sure in a mathematically rigorous setting, we would write $b^* \to a^*$ instead for the last line, where b^* is the dual space. Since this is functional programming though, we canonically associate the dual space with the standard space using the standard real inner product.

8 September 29th, 2020

Today, we have an ethics lecture. Zach Gabor (G5 in Philosophy) from the Embedded EthiCS team will present and lead the discussion on ethics in Interpretable AI.

8.1 Interpretable Machine Learning [DVK17]

A fundamental question in the machine learning ethics literature is the *right to explanation*. First, we should define what it means to have an explanation. This idea primarily comes from the psychology literature.

Definition 8.1 (Explanation)**.** An *explanation* is "the currency in which we exchange beliefs" [Lom06]. This begs the questions: what constitutes an explanation, what makes some explanations better than others, and how are explanations and sought?

Oftentimes, people care about explanation for the purpose of eliminating algorithmic *unfairness* and avoiding discrimination. Extracting an explanation from a model is one way that we can discharge worries about discrimination, but it is not necessarily the only way, and does not *always* work. Zach suggests that we separate the concepts of (1) right to explanation and (2) avoiding discrimination.

Example 8.2 (Temperamental banker)**.** Fairness may extend beyond preventing discrimination against protected groups. Consider the *temperamental banker*, who flips a coin whenever he decides to grant a loan. This is not discriminatory toward protected groups, but it may seem somewhat arbitrary. As a rejected applicant, you might find grounds to file a complaint, as the loan-filing process is *unreliable*.

We can attempt to define basic fairness, in some sense.

Definition 8.3 ([Sun06])**.** Two competing notions of *procedural fairness* are:

1. Fairness as universal, consistent application of rules (e.g., rules for mandatory death penalty cannot be universally applied due to individual *ad hoc* judgments).
2. Fairness as individual consideration (e.g., quota or points-based affirmative action are bad, as they reduce individuals to data points).

Both of these definitions have their own advantages, and are not inherently incompatible, but often compete with each other in practice. Rule-bound decisions potentially reduce work, are predictable, and avoid humiliating individuals with excessive scrutiny. On the other hand, individualized decisions allow for discretion in exceptional cases.

Note. Much of this discussion is relevant to the process of college admissions, which has historically been done behind closed doors due to outside criticism. We would ideally like to have both fair consideration *and* transparency.

Machine learning models may fail at individual considerations. Exceptional cases typically occur less often in the training data. One relatable example is *AlphaGo*, which does not perform well in less-explored parts of the game tree. Tradeoffs in the algorithm space include:

- Global, versus local explanations.
- No human intervention, versus abundant human intervention.

8.2 Discussion Time: Fairness

Prof. Amin notes that rule-bound and individual notions may not be disjoint in general, as you can have a model that is rule-bound yet takes individual notions as input. This may be the dominant model in college admissions processes.

Andrew Kim mentions a counter point to Example 8.2 if the use of probability can influence the fairness of a model. For example, if a lottery system is used in some subset of the population, is that fair? Does randomness inherently affect the fairness value of a model, or is it okay to have decisions that rely on some stochasticity (e.g., the lottery), as long as it is explicit? Zach labels this an open question. Another question of interest is whether *fairness itself*, or the *appearance of fairness*, is more important.

One major problem with rule-bound decisions is that they allow people to game the system. People will simply optimize for satisfying the algorithm, rather than taking an actionable improvement on their life or utility. In particular, if you do not have the ϵ-ball robustness property (common for image classifiers and other high-dimensional inputs), then people would be able to manually "gradient descent" their inputs on the model, resulting in a different outcome with minimal change.

8.3 ML Ethics Workshop at Princeton

Here's a related workshop with Harvard Philosophy people, linked by Mark Goldstein.

The Ethics of Algorithmic Decision-Making in Democratic Institutions

> The normative implications of algorithmic decision-making are currently generating lively debate in multiple academic disciplines, such as applied ethics and computer science (especially in the emerging subdiscipline of *FAT*ML*: fairness, accountability and transparency in machine learning). By contrast, political philosophers—including democratic theorists—have engaged with the subject of algorithmic decision-making much less. This is surprising, given that algorithmic tools, as they are used here and now, seem to raise complex new questions for long-standing debates in political philosophy: on equality, justice, discrimination, autonomy, accountability, responsibility, and power. As democratic institutions themselves increasingly rely on algorithmic tools in criminal justice, in law enforcement, and in decision-making about the allocation of resources and benefits, these questions become even more pressing. This workshop aims to bring together political philosophers interested in exploring the democratic implications of algorithmic tools.

9 October 1st, 2020

Today, Anastasiya Kravchuk-Kirilyuk will present the paper "Probabilistic Verification of Fairness Properties via Concentration" [BZSL19].

9.1 Probabilistic Verification of Fairness

In life, programs will need to be evaluated according to their *fairness*. We would like to have an algorithm that automates testing for fairness properties. The algorithm we introduce will be called *VeriFair* (probably pun intended?). This uses the following techniques:

- Random sampling from the population model.
- Adaptive concentration inequalities to determine when to stop sampling.
- Probabilistic bounds on guarantees.

First, how do we define fairness anyway? Suppose that we have a simple hiring model based on years of experience and rank, but the population model generates males that disproportionately have a higher number of years of experience. This is shown in Fig. 13.

```
def offer_job(col_rank, years_exp)
   if col_rank <= 5:
      return true
   elif years_exp > 5:
      return true
   else:
      return false
```

```
def population_model():
   is_male ~ bernoulli(0.5)
   col_rank ~ normal(25, 10)
   if is_male:
      years_exp ~ normal(15, 5)
   else:
      years_exp ~ normal(10, 5)
   return col_rank, years_exp
```

Figure 13: Potential population and hiring model for a company.

The notion of fairness here is called *demographic parity*.

Definition 9.1 (Demographic parity)**.** We say that a classifier has *demographic parity* when the hiring rate for candidates of the minority group is at least **80%** the hiring rate for the majority group.[6] In other words,

$$Y_{\text{job}} = \left(\frac{\mu_{\text{male}}}{\mu_{\text{female}}} \geq 0.8 \right),$$

where $\mu_a = \Pr(\text{offer} = 1 \mid \text{gender} = a)$ for each $a \in \{\text{male}, \text{female}\}$.

Note that this definition may be slightly controversial (versus purely meritocratic decisions), but it is certainly a starting point for a definition of fairness, so we will fix it for the sake of discussion. To estimate Y_{job}, we simply draw a list of samples $V_{a,1}, \ldots, V_{a,n}$ from each subpopulation a, in the conditional population distribution, and we compute some $\hat{\mu}_a$ estimated hiring ratio for each subpopulation a.

Using adaptive concentration inequalities, we can pick some parameter δ_a so that $\Pr(|\hat{\mu}_a - \mu_a| \geq \epsilon) \leq \delta_a$. After that, we can use standard union bounds to show that $\Pr(\hat{Y}_{\text{job}} = Y_{\text{job}}) \geq 1 - \gamma$ for

[6] In general, we can have c-parity for any $c \in [0, 1]$, by replacing 80% with $1 - c$.

$$\frac{\mu_Z : (E, \varepsilon, \delta) \in \Gamma}{\Gamma \vdash \mu_Z : (E, \varepsilon, \delta)} \text{ (random variable)} \qquad \frac{c \in \mathbb{R}}{\Gamma \vdash (c, 0, 0)} \text{ (constant)} \qquad \frac{\Gamma \vdash X : (E, \varepsilon, \delta),\ \Gamma \vdash X' : (E', \varepsilon', \delta')}{\Gamma \vdash X + X' : (E + E', \varepsilon + \varepsilon', \delta + \delta')} \text{ (sum)}$$

$$\frac{\Gamma \vdash X : (E, \varepsilon, \delta)}{\Gamma \vdash -X : (-E, \varepsilon, \delta)} \text{ (negative)} \qquad \frac{\Gamma \vdash X : (E, \varepsilon, \delta),\ |E| > \varepsilon}{\Gamma \vdash X^{-1} : (E^{-1}, \frac{\varepsilon}{|E| \cdot (|E| - \varepsilon)}, \delta)} \text{ (inverse)}$$

$$\frac{\Gamma \vdash X : (E, \varepsilon, \delta),\ \Gamma \vdash X' : (E', \varepsilon', \delta')}{\Gamma \vdash X \cdot X' : (E \cdot E', |E| \cdot \varepsilon' + |E'| \cdot \varepsilon + \varepsilon \cdot \varepsilon', \delta + \delta')} \text{ (product)}$$

$$\frac{\Gamma \vdash X : (E, \varepsilon, \delta),\ E - \varepsilon \geq 0}{\Gamma \vdash X \geq 0 : (\text{true}, \delta)} \text{ (inequality true)} \qquad \frac{\Gamma \vdash X : (E, \varepsilon, \delta),\ E + \varepsilon < 0}{\Gamma \vdash X \geq 0 : (\text{false}, \delta)} \text{ (inequality false)}$$

$$\frac{\Gamma \vdash Y : (I, \gamma),\ \Gamma \vdash Y' : (I', \gamma')}{\Gamma \vdash Y \wedge Y' : (I \wedge I', \gamma + \gamma')} \text{ (and)} \qquad \frac{\Gamma \vdash Y : (I, \gamma),\ \Gamma \vdash Y' : (I', \gamma')}{\Gamma \vdash Y \vee Y' : (I \vee I', \gamma + \gamma')} \text{ (or)} \qquad \frac{\Gamma \vdash Y : (I, \gamma)}{\Gamma \vdash \neg Y : (\neg I, \gamma)} \text{ (not)}$$

Figure 14: Inference rules used for concentration inequalities on *expressions*.

some chosen error parameter γ. This should remind you very much of a Probably Approximately Correct (PAC) learning approach.

This is a pretty basic approach to verifying fairness, just standard concentration inequalities and verification by bounding. However, this is an OOPSLA paper, not a COLT paper. The bulk of the paper is about applying this in a smart way to verification of fairness. They also write their algorithm in the framework of logical inference rules, as is common in the programming language literature. A derivation of the form $\Gamma \vdash X : (E, \epsilon, \delta)$ is a statement about the marginal distribution of X, that $\Pr(|X - E| \geq \epsilon) \leq \delta$. Then, the full set of inference rules is shown in Fig. 14; they're easily proven by taking union bounds.

From here on, the algorithm is very simple. The only requirement is that we are able to sample from the population distribution, and that the fairness property being estimated (not necessarily just demographic parity) is a function of the means of Bernoulli random variables (e.g., whether a candidate is hired). Then we can use the concentration expression grammar above to verify concentration properties for general expressions involving arithmetic and inequalities.

9.2 Adaptive Concentration Inequalities

We've seen in the last section how to integrate concentration inequalities in a general framework to talk about concentration for expressions. Now let's briefly discuss how the concentration itself works. We can't use standard Chernoff/Hoeffding/Azuma bounds because the number of samples taken is a random variable. If we only draw samples until reaching a desired error margin, that would violate independence properties (see: p-hacking).

Instead, we can use *adaptive* concentration inequalities, which allow us to draw variables until reaching our desired error margin, in a mathematically rigorous way. This framework was introduced in NeurIPS 2016, as a way of solving precisely the task of sequential decision problems.

Theorem 9.2 (Adaptive concentration, [ZZSE16])**.** *Given a Bernoulli random variable Z with distribution $P_{\mathcal{Z}}$, let $\{Z_i \sim P_{\mathcal{Z}}\}_{i \in \mathbb{N}}$ be i.i.d. random samples of Z, and let*

$$\hat{\mu}_Z^{(n)} = \frac{1}{n} \sum_{i=1}^{n} Z_i.$$

Let J be a random variable on $\mathbb{N} \cup \{\infty\}$ such that $\Pr(J < \infty) = 1$, and let

$$\epsilon(\delta, n) = \sqrt{\frac{\frac{3}{5} \cdot \log(\log_{11/10} n + 1) + \frac{5}{9} \cdot \log(24/\delta)}{n}}.$$

Then, given any $\delta > 0$, we have

$$\Pr\Big(|\hat{\mu}_Z^{(J)} - \mu_Z| \leq \epsilon(\delta, J)\Big) \geq 1 - \delta.$$

So essentially, this paper contributes the application of adaptive concentration, along with a nicely written framework for talking about concentration of expressions. It then uses it to examine the hot question of fairness and bound some large real-world RNN models.

Note. In practice, this is just a glorified mathematical version of the following test: look at how many men and women the model classifies as 1 or 0, then compare their marginal distributions. However, it's nice that there's some mathematical rigor in this Bayesian concentration approach, versus the standard confidence intervals that we often see in the literature.

10 October 6th, 2020

Today, Alex Gu (MIT '22) will present a talk on the paper "Program Synthesis with Pragmatic Communication" [PEK+20]. This is a recent paper from Josh Tenenbaum's lab.

10.1 Pragmatic Communication for Synthesis [PEK+20]

The abstract is introduced as follows. Program synthesis techniques construct or infer programs from user-provided specifications, such as input-output examples. Yet most specifications, especially those given by end-users, leave the synthesis problem *radically ill-posed*, because many programs may simultaneously satisfy the specification. Prior work resolves this ambiguity by using various inductive biases, such as a preference for simpler programs. This work introduces a new *inductive bias* derived by modeling the program synthesis task as **rational communication**, drawing insights from recursive reasoning models of pragmatics. Given a specification, we score a candidate program both on its consistency with the specification, and also whether a rational speaker would chose this particular specification to communicate that program.

Roughly speaking, input/output examples are never good enough to fully specify a program, as there will always be many possible programs consistent with those examples. This is similar to how in communication, language can often be ambiguous. One way of resolving ambiguity in language is to use some *pragmatic* interpretation of the language. This can be represented as a language model representing probabilities and syntax that you have already seen.

Question: Joey brings up, why is this pragmatic model distinct from existing biases, like leaning towards the simplicity of the generated program? It would be interesting to find an example where pragmatics do well, but choosing the shortest program doesn't do as well.

How might this idea be related to compression, or information theory? It seems like a pragmatic distribution is in some sense related to creating an encoder-decoder pair, given by distributions $P_S(D \mid h)$ and $P_L(h \mid D)$, for the encoder and decoder respectively. This is interesting because we add some extra layer of communication interface to the synthesis algorithm. Shashank brings up [VT20], a hot paper on the topic probing through minimum description length.

Incrementally Pragmatic Speaker S_1. We now build a pragmatic speaker S_1 recursively from L_0. Here, rather than treating D as an unordered set, we view it as an ordered *sequence* of examples, and models the speaker's generation of D incrementally, similar to autoregressive sequence generation in language modeling [26]. Let $D = u^1 \dots u^k$, then:

$$P_{S_1}(D|h) = P_{S_1}(u_1, \dots, u_k|h) = P_S(u_1|h)P_S(u_2|h, u_1) \dots P(u_k|h, u_1 \dots u_{k-1}) \quad (2)$$

where the incremental probability $P_S(u_i|h, u_1, \dots, u_{i-1})$ is defined recursively with L_0:

$$P_S(u_i|h, u_{1\dots i-1}) \propto P_{L_0}(h|u_{1\dots i}), \quad P_S(u_i|h, u_{1\dots i-1}) = \frac{P_{L_0}(h|u_1, \dots, u_i)}{\sum_{u_i'} P_{L_0}(h|u_1, \dots, u_i')} \quad (3)$$

Applying this reasoning to our example in Figure 2, we see that $P_{S_1}(u_2, u_4|h_5)$ is:

$$P_S(u_2|h_5)P_S(u_4|h_5, u_2) = \frac{P_{L_0}(h_5|u_2)}{\sum_{u'} P_{L_0}(h_5|u')} \frac{P_{L_0}(h_5|u_2, u_4)}{\sum_{u''} P_{L_0}(h_5|u_2, u'')} = 0.25 * 0.3 = 0.075 \quad (4)$$

Informative Listener L_1. Finally, we construct an informative listener L_1 which recursively reasons about the informative speaker S_1:

$$P_{L_1}(h|D) \propto P_{S_1}(D|h), \quad P_{L_1}(h|D) = \frac{P_{S_1}(D|h)}{\sum_{h'} P_{S_1}(D|h')} \quad (5)$$

In our example, $P_{L_1}(h_5|u_{2,4}) \approx 0.31$, $P_{L_1}(h_2|u_{2,4}) \approx 0.28$, $P_{L_1}(h_3|u_{2,4}) \approx 0.19$, $P_{L_1}(h_6|u_{2,4}) \approx 0.21$. As we can see, the intended concept h_5 is ranked first, in contrast to the uninformative listener L_0.

Figure 15: Details of the pragmatic synthesis algorithm.

Figure 15 shows how the encoding and decoding algorithm works, at a high level. It is fairly simple but inefficient, taking $O(|\mathcal{H}| \cdot |\mathcal{U}|)$ time, where $\mathcal{H}$ is the hypothesis class, and $\mathcal{U}$ is the set of possible input/output examples. The primary idea is that the *pragmatic speaker* generates a sequence of examples using an autoregressive language model (e.g., n-gram Markov, RNN, Transformer) on the hypothesis h. Then, the informative listener receives these examples on a one-way channel and uses each example to do a Bayesian update on the likeliness of each hypothesis.

10.2 Experiments on Pragmatic Synthesis

For what it's worth, the primary contribution of this paper is operational, by taking the idea of pragmatics from the cognitive science literature and applying it to computational learning theory. This is interesting from a theoretical perspective, but it helps to see real examples where it actually performs well. The paper gives the example of learning some grids of cells with structure, where the speaker tries to produce examples that reflect the borders of the grids. This allows the listener to narrow down the set of valid hypotheses as quickly as possible.

From breakout discussions with Joey: "There's a trend to take a sexy idea in cognitive science and try to do cool stuff with it in an applied setting." But it's unclear what an alternative formulation of this approach might look like, such as variational inference or other techniques.

Question: What prevents the speaker and listener from colluding on a "secret code" that enables perfect compression, by assigning each subset of examples to a different hypothesis, but doesn't really actually make sense from the generalization, or learning perspective?

For practical applications, https://pyro.ai/examples/RSA-implicature.html shows some smart sampling methods or variational inference to do pragmatics research in a computationally efficient manner. We're curious about what the connections with online compression are—is this really the same framework in disguise of different wording? I feel like this paper really introduces more questions than it actually answers.

11 October 8th, 2020

Today, Romil Sirohi will present the paper "Bootstrap Learning Via Modular Concept Discovery" [MAT13]. This is yet another paper from Josh Tenenbaum, which introduces the *E.C.* algorithm.

11.1 Modular Concept Learning [MAT13]

The motivating example behind this paper is learning a calculator. How can we get a computer to intelligently design an electronic circuit for a calculator, without domain knowledge? Such a circuit would be immensely intricate, and one possible approach would be to start from simple, reusable components based on electronics (e.g., gates, memory cells, latches, etc.), and combining those components together in a *modular* way. We call this process of developing simple components *bootstrapping*: the creation of knowledge without domain knowledge.

Applications shown in this paper will use a bootstrapping algorithm to develop new approaches to symbolic regression (for polynomials), as well as NAND programs. General search algorithms always exploit some topological structure in the search space, but in this case, we will search based on the topology of a DSL grammar. Then, our search algorithm will use this DSL to generate some likelihoods for each program in the DSL, which are then used to build a library of programs.

> 1) Can an AI system discover the structure latent in the solutions to multiple related problems and thereby bootstrap effective exploration of the search space?
>
> 2) Can discovering modular program components transform a problem from one in which search is intractable into a problem in which it becomes increasingly feasible with experience?

The algorithm introduced in the paper, called *E.C.*, is an iterative approach with several steps. It starts with a set of tasks to solve.[7] Then, the algorithm has a distribution $\mathcal{D}$ over a language $\mathcal{L}$, while keeping a frontier of size N of the most likely programs. We say that a task is *hit* by a program if that program solves all input/output examples exactly. At each iteration:

1. **Exploration:** Create a new frontier, enumerate the N most probable expressions from $\mathcal{D}$.
2. **Compression:** Uses the hit tasks in the frontier to create a new distribution $\mathcal{D} \mapsto \mathcal{D}'$.

Those are the high-level ideas of the algorithm, maintaining a distribution and doing some best-first enumeration to update the distribution iteratively, similar to genetic programming.

11.2 Details of the E.C. Algorithm

To learn general functional programs, we will use *simply-typed combinatory logic*. The reason for using combinators (shown in Fig. 16), versus lambda calculus, is that combinatory logic does not have any free variables. This makes it simpler to construct valid trees of programs. In general, combinatory logic is equivalent to lambda calculus.

This brings up a couple of questions, from Joey, Shashank, myself, and others:

- If the combinatory system is typed, we can't express the Y combinator and can't get recursion. This might mean that the language is not Turing complete.

[7] Note that although we supposedly do not need domain-specific knowledge, these tasks are actually domain-specific questions in some sense. By giving the computer a set of tasks, we are telling the computer that solving these tasks might be useful. It's almost like giving a guided tutorial towards incrementally assembling complex components.

$$\begin{aligned}
\mathbf{I}\ x &\to x && \text{(identity)} && (1)\\
\mathbf{S}\ f\ g\ x &\to (f\ x)\ (g\ x) && && (2)\\
\mathbf{C}\ f\ g\ x &\to (f\ x)\ g && && (3)\\
\mathbf{B}\ f\ g\ x &\to f\ (g\ x) && \text{(composition)} && (4)
\end{aligned}$$

Figure 16: Combinators in the simply-typed system.

- Combinatory logic might be a much longer representation of some simple programs, and it might also be less readable than a high-level language.
- If the language is not Turing complete, it can be guaranteed to halt, which might allow the symbolic interpreter to be simpler.

Back to the main topic of the algorithm. In addition to the parametric combinators shown above, we also introduce primitive combinators for arithmetic and other operations. Then we construct our trees in a top-down fashion. Each leaf *requests* some type. We then run a *unification algorithm* to determine which combinators or atoms could be inserted at that leaf node.

Our stochastic grammar over the space of programs is defined through conditional distributions. An expression $e \in \mathcal{L}$ has probability equal to

$$p(e) = \prod_{c \in C_e} p(c \mid \tau(c)),$$

where $p(c \mid \tau(c))$ is the probability of choosing primitive c when the requesting type is $\tau(c)$. On the other hand, the conditional probability for each combinator c_n is

$$p(c_n \mid \tau(c_n)) \propto \frac{p_n}{\sum_{c_j \in C_{\tau(c_n)}} p_j}.$$

To make everything converge for binary trees, we set the constant of proportionality to be $\frac{1}{4}$. Otherwise, it can be shown that the probabilities blow up in a recurrent manner.

Now that the distribution is defined, we still need an algorithm to actually conduct best-first enumeration of the program space. To do this, we formulate a program as iterative exploration of an AND/OR tree. Some leaves have multiple options (type unification), which would create an OR node, while each OR node then has an AND node referring to the types required by each input to a function composition, `'a -> T` and `'a`.

The next hurdle of the algorithm is to find the *most compressive* set of solutions. Briefly, when there are multiple programs that satisfy the same tasks, the algorithm tries to only pick one or more algorithms for each task that minimize the total number of *distinct subtrees* in all programs. They mention that an exact solution might take exponential time, but for their purposes, a greedy/heuristic approach worked reasonably well and in polynomial time.

Finally, in the next step after compression, the algorithm tries to create new primitive combinators based on the discovered modular concepts. After creating new primitives, however, we need to update our conditional probability rules! We use a variant of the Nevill-Manning algorithm for this, which constructs a context-free grammar from a set of examples. Note however that we don't

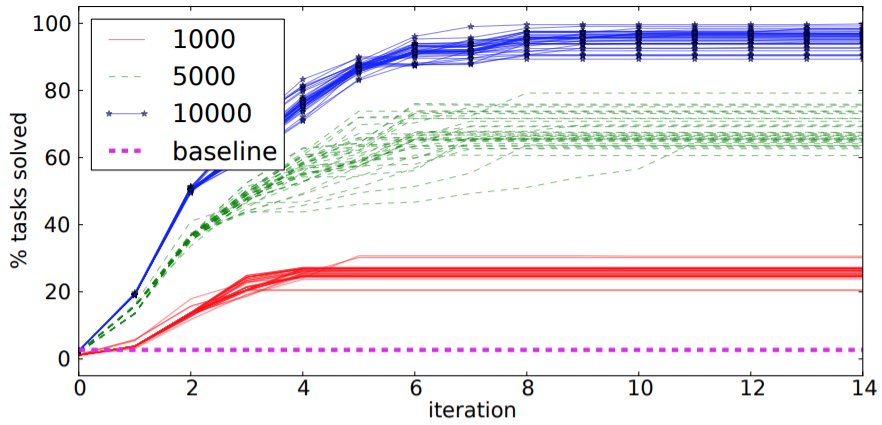


Figure 2: Learning curves as a function of frontier size. As frontier size is increased, curves plateau closer to 100% performance. A baseline search over 150000 expressions only hits 3% of the tasks (dashed pink line).

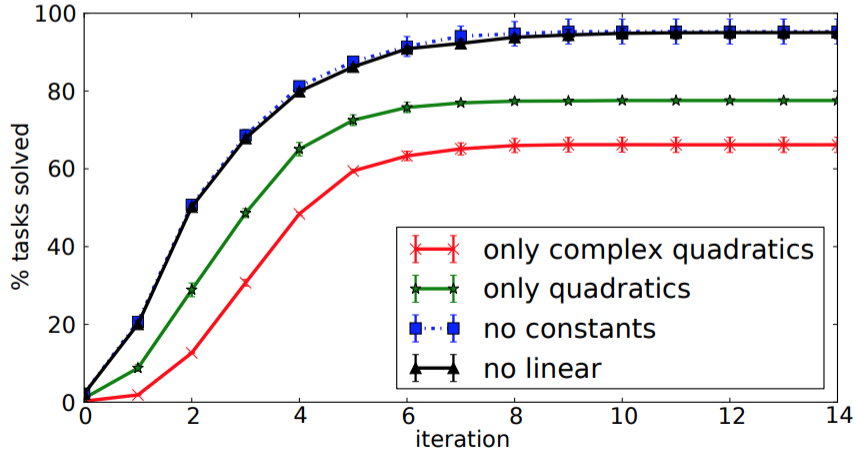


Figure 3: How do different task sets affect learning curves? Learning curves at a frontier size of 10000 for different task sets.

Figure 17: Results of running E.C. on symbolic regression.

explicitly use the previous probabilities anywhere (not a Bayesian update); this is only implicitly reflected in how the examples were initially generated.

Results for symbolic regression tasks are shown in Fig. 17. The algorithm was also run for NAND circuits, for which it discovered several useful Boolean primitives, and the paper also demonstrates visually that it is somewhat good at learning the distribution of circuits in the input.

Joey mentions some interesting work on fragment grammars, a research topic in linguistics that also uses a similar idea of bootstrapping distributions iteratively [OSTG11]. These are both ancestors to the *DreamCoder* paper [EWN+20], which we will discuss in a later class.

12 October 13th, 2020

Today we have a guest lecture from Will Byrd, who will talk about his work on *miniKanren* with Nada Amin, i.e., program synthesis through a relational interpreter. Will makes the point that if anything he says is wrong or falsifiable, please let him know, as engineering is about making tradeoffs. Daniel Friedman is also present, a famous person in the Lisp / Scheme community.

12.1 Relational Programming [Byr10]

A lot of the basis of program synthesis techniques comes from the famous textbook, "Structure and Interpretation of Computer Programs" [AS96]. Lots of interesting research came from the 1970s, where many researchers thought they were on the cusp of making truly intelligent systems. Although the were far off (the problem is very hard), many interesting ideas come from research in this time period.

Will's area of research is relational programming, and his primary contribution has been the miniKanren language [Byr10]. Generally, relational programming comes in several forms. We have the classical SQL, Datalog, and Prolog hierarchy, in increasing order of power. The general idea of all of these languages is that rather than having functions, we have *relations* between ground members. These relations are reversible and query-able.

```
> (+ 1 2)
3
> (+ 3 5)
5
;; + int x int -> int
;; +o: int x int x int
(+o 3 5 8)
;; Table representing addition:
(+o x y z)
;; x y z
;; 0 0 0   (0 0 0)
;; ...
;; 3 5 8   (3 5 8)
;; ...
(+o 3 ? 8) => ? = 5                    ; Created subtraction!
(+o ?1 ?2 8) => (0 8), ..., (4 4), ...  ; Multiple answers
(+o ?1 ?2 ?3) => ...                   ; Infinitely many answers
(+o 5 ? 3) => nil                      ; No answers
```

See the example above for a conceptual introduction to relational programming, written in the functional Scheme language. Note that if we want to have relations such as `+o`, it's essential that we are able to effectively deal with *infinite databases*, which require some notion of laziness. This is where the fundamental usefulness of miniKanren comes in.

Note. Will makes the point that miniKanren is an extremely small core language, and it's therefore very easy to implement in any host language as an embeddded DSL. For example, see μKanren for a helpful 50-line reference implementation in Scheme.

Since miniKanren is a flexible embedded language, it needs a way to interface with the host. Generally, there are only three points of interest in the interface: the functions `==`, `conde`, and `fresh`. There are also two functions, `run` and `run*`, for actually executing the program.

```
> (run 1 (q) (== 5 5))  ; 5 == 5 for all values of q
(_.0)
> (run 1 (q) (== 5 6))  ; 5 == 6 for no values of q
()
> (run 1 (q)
    (== (list 5 5)
        (list 5 q)))    ; (5 5) == (5 q) when q = 5
(5)
> (run 2 (q)
    (== (list 5 5)
        (list 5 q)))
(5)
> (run* (q)
    (== (list 5 5)
        (list 5 q)))    ; "proving" q = 5 is the only solution
(5)
> (run 2 (q)
    (conde
      ((== q 5))
      ((== q 6))))      ; modeling disjunction
(5 6)
> (run* (q)
    (fresh (x y z)
      (== (list y z) x)
      (== y 5)
      (== z 6)))        ; create a non-capturing scope for x, y, z
(_.0)
> (run* (x y z)
    (== (list y z) x)
    (== y 5)
    (== z 6))           ; this time, capture the values
((5 6) 5 6)
```

That's all there is for the core language of miniKanren! There's just a couple of simple operators and a way of interfacing with the host. Now let's see some more advanced examples, integrating the core features with external procedures as well. For example, recall the built-in Scheme function `append`, which concatenates two lists.

```
> (append '(a b c) '(d e))  ; built-in Scheme append
(a b c d e)
```

This is a very classic function, and it's one of the first functions any functional programmer might learn to implement. Now let's see how we might translate this function into a relational version with miniKanren.

```
> (define appendo
    (lambda (l s l+s)
      (conde
        ((== '() l) (== s l+s))
        ((fresh (first rest res)
           (== (cons first rest) l)
           (== (cons first res) l+s)
           (appendo rest s res))))))
> (run 1 (q) (appendo '(a b c) '(d e) q))
((a b c d e))
```

You can see how we can constructively build up more complex functions using this approach!

12.2 Synthesis with a Relational Interpreter

Now let's see how miniKanren can has relevance within the program synthesis research area. Recall that we can search for all satisfiable assignments for a given expression with miniKanren, although such a search might run forever in the worst case.

```
> (run* (a b)
    (appendo a b '(1 2 3 4 5)))
((1 2 3 4 5) ()
 (1 2 3 4) (5)
 (1 2 3) (4 5)
 (1 2) (3 4 5)
 (1) (2 3 4 5)
 () (1 2 3 4 5))
```

Fair enough, this makes sense. However, recall that Lisp is a *homoiconic* language, meaning that we can store and manipulate code as data. Therefore, what if we searched over parameterized bindings for *programs*, rather than just data? To accomplish this, we will use `evalo`, which is a relation implementing `eval` for a subset of the Scheme language.

```
> (eval '(+ 3 5))
8
> (run 1 (q) (evalo '(list) q))
(())
> (run 1 (q) (evalo '(list 'a 'b 'c) q))
((a b c))
> (run 1 (q) (evalo '(list 'I 'love 'scheme) q))
((I love scheme))
> (run 10 (q) (evalo q '(I love scheme)))
('(I love scheme)
  ...)  ; various other complex lambdas
```

This is already really interesting! By parameterizing over the program rather than the data, we can let `run` iterate over some of the infinitely many programs that might output a value. It turns

out that we can implement pretty large subsets of scheme in `evalo`. For example, see the program below.

```
> (run 1 (val)
    (evalo
     `(letrec ((append
                (lambda (l s)
                  (if (null? l)
                      s
                      (cons (car l)
                            (append cdr l) s)))))
        (append '(a b c) '(d e)))
     val))
((a b c d e))

> (run 1 (e)
    (evalo
     `(letrec ((append
                (lambda (l s)
                  (if (null? l)
                      ,e
                      (cons (car l)
                            (append cdr l) s)))))
        (append '(a b c) '(d e)))
     '(a b c d e)))
s
```

Note that in the second example above, we actually managed to infer the code point that `e` replaces inside the function above! This shows that miniKanren is actually able to do some reasonable interactive inference, by filling in missing code at various parts of a program. The fundamental reason that lets this work is that we encode the *operational semantics* of a subset of Scheme into a miniKanren program.

This can be optimized and eventually refined to create Barliman, an open-source relational program synthesis tool. You input a set of input/output tests, some potentially including *gensyms*, and the program outputs a best guess for the program that produces those tests. See Fig. 18 for an example of this.

Beyond this very brief introduction, a lot of the "secret sauce" behind miniKanren for synthesis is the search algorithm. It neither uses a depth-first or breadth-first approach like Prolog or Datalog, but rather a complex algorithm based on a data structure called *MonadT*. Right now, a lot of recent work involves trying to find ways of combining different AI systems, such as neural network-guided search, with relational programming. Further reading on this topic includes a book from MIT Press called *The Reasoned Schemer*, and Clojure's core.logic library.

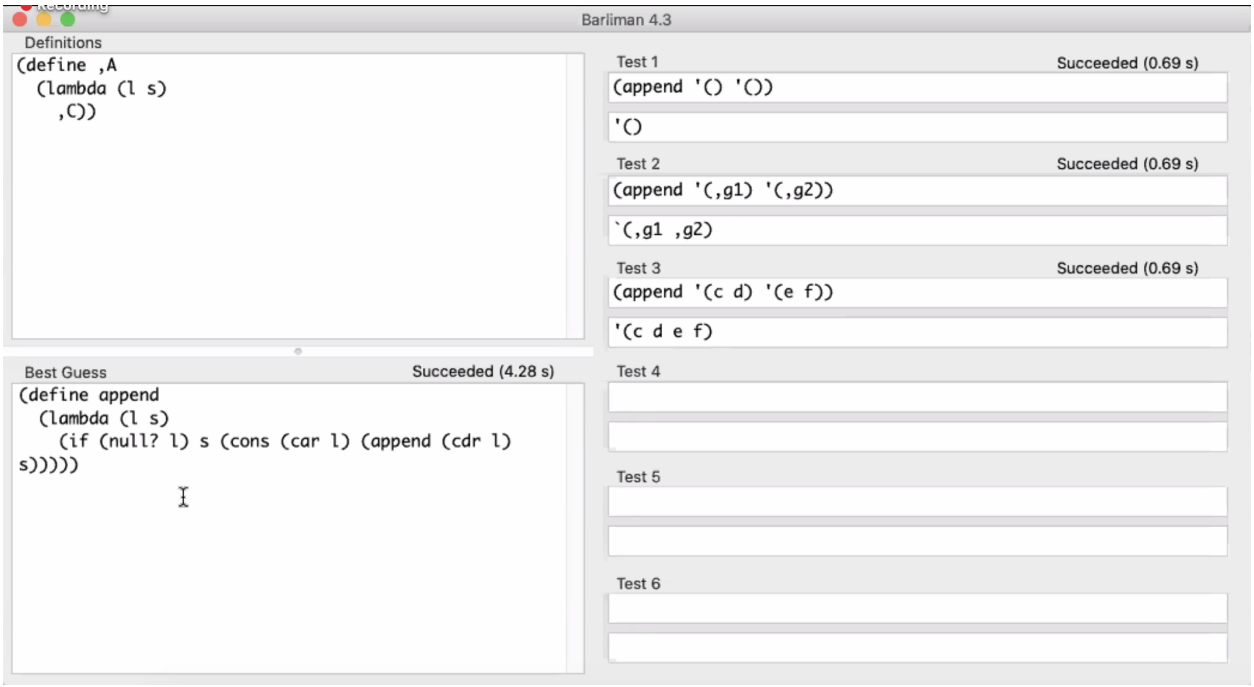


Figure 18: Example of Barliman inference for the append function.

13 October 15th, 2020

Today, I (Eric Zhang) will present the paper "DeepCoder: Learning to Write Programs" [BGB+16]. I'm adding some notes to this document retroactively, to outline my presentation and some interesting comments that came up.

13.1 Learning-Augmented Algorithms [KBC+18]

Before we discuss program synthesis as a whole, we first briefly discuss a broader topic that has deep connections with the current paper: learning-augmented algorithms. This is a very recent topic of interest (c.f., 6.890, or the STOC'20 Workshop on Algorithms with Predictions), and it presents a very similar approach to that of DeepCoder.

Definition 13.1 (Learning-augmented algorithm)**.** A *learning-augmented algorithm* combines existing, functional approaches with a black-box machine learning model (such as a neural network). Under mild assumptions about the competence of the machine learning predictions, we can achieve significant speedups or improvements on the non-augmented algorithm.

We will discuss two examples. First, consider that of binary search.

Example 13.2 (Augmented binary search)**.** Suppose that you have a sorted array of length n and a query that asks whether a number x appears in that array. By applying classical binary search (repeated halving), we can easily solve the task in worst-case $O(\log n)$ time, by finding the index i at which x occurs in the array, or if it is not found, the index where it would be inserted.

However, now suppose that you had a black-box machine learning model $f : x \mapsto j$ that makes a prediction j about the index that x occurs. Let $t = |i - j|$ be the error between the prediction and the actual result. Then, if you run a modified binary search starting from index j, you can provably achieve average-case $O(\log t)$ runtime, which is better than $O(\log n)$.

Note that in the algorithm above, the worst-case runtime, even when your classifier is absolutely terrible ($t = n - 1$), is still $O(\log n)$. In other words, it is within a constant factor of the original algorithm. This is a really nice property because it means, in some sense, that our algorithm can *only get better* asymptotically by adding the learned part.

Example 13.3 (Learned Bloom filter [KBC+18])**.** The most famous augmented algorithm is the Bloom filter. Bloom filters are a classic data structure that allow you to conduct set membership queries while storing only a minimal amount of data, with the tradeoff of some small probability of false positives. This makes errors from a black-box classifier relatively acceptable as well.

We put a neural network in front of the Bloom filter, as shown in Fig. 19. By accepting some small increase in false positives, we can dramatically reduce the load (and thus memory usage) of the underlying Bloom filter, improving its performance.

Hopefully this talk of learning-augmented algorithms is inspiring. Whereas many approaches we've seen so far in this seminar have been end-to-end deep learning, backpropagating through the entire system, augmented algorithms give a fresh new take on the problem. Machine learning can be used as a black box to give hints to existing, successful domain algorithms.

13.2 Learning Inductive Program Synthesis (LIPS) [BGB+16]

This paper tries to augment existing approaches for *inductive program synthesis* with deep learning. Here, the word "inductive" means inference, in that we are inferring the source code of programs from a set of input-output examples. Call our set of examples E and our program P.

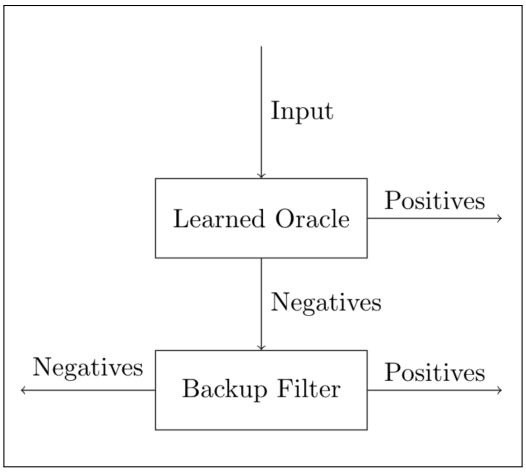


Figure 19: A learned Bloom filter.

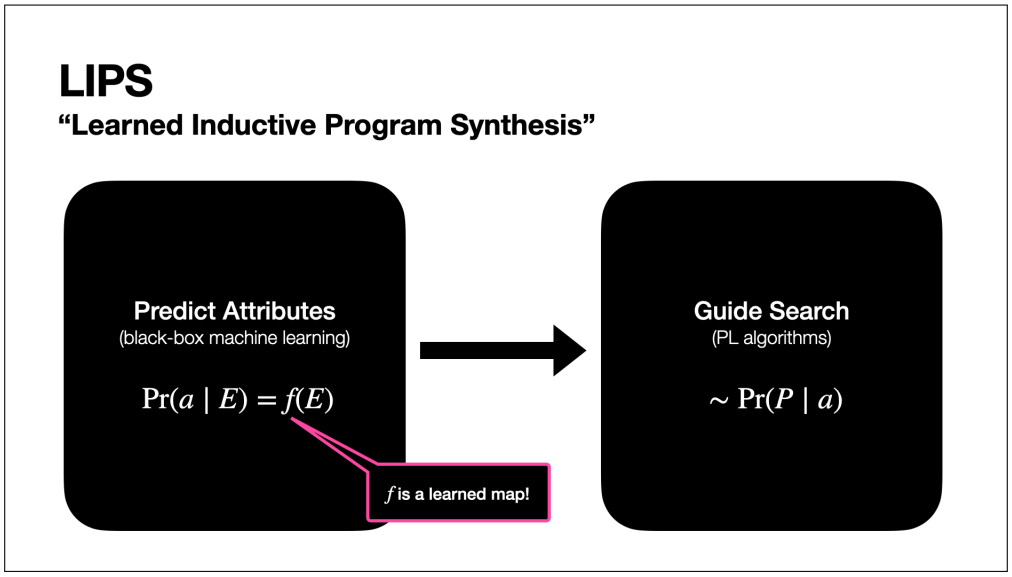


Figure 20: Diagram showing the two parts of LIPS.

Definition 13.4 (Attributes)**.** We define an *attribute function* $\mathcal{A} : \mathcal{P} \to \mathbb{R}^C$, mapping each program in the DSL into a vector of real numbers that summarizes it. In this case, $C = 34$ is the number of functions in the DSL, and our attributes are just a 0–1 encoding of whether each function is present in the program $P \in \mathcal{P}$.

The high-level approach of LIPS is shown in Fig. 20. First, we start with our input-output examples E and use a learned function f to produce a prediction of the attribute vector a. This is reflected in the probability distribution $f(E) \sim \Pr(a \mid E)$. Then, we use the predicted attribute vector a as an input to the program synthesis algorithm. Consider the example pair shown below.

```
Input:
  [-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11]
Output:
  [-12, -20, -32, -36, -68]
```

As a human, we can immediately make several inferences about the attributes of the program P that likely produced this input-output example:

- All the outputs are even, so we might have the `map`, `filter`, or `(*n)` functions.
- All the outputs are sorted, so we might have the `sort`, or `reverse` functions.

Our goal in LIPS will be to *model* this human intuition by using deep learning on E. The advantage of this approach, over an end-to-end neural translation like in *R3NN* [PMS+16], is that we only backpropagate losses for the initial perceptual part of the algorithm, which produces an interpretable intermediate vector a. Since this is so simple, we can fairly easily plug it into virtually any existing search algorithm, even if they are not differentiable. Shown below is an example data file that might be passed into the deep learning model.

```
[
  {
    "program": "LIST|LIST|COUNT,>0,0|ZIPWITH,+,1,0|ACCESS,2,3",
    "examples": [
      {
        "inputs": [
          [-46, -23, -78, 10],
          [125, 105, -69]
        ],
        "output": 82
      },
      {
        "inputs": [
          [90, 103, -57, 13, -45, 28, -30, 68, -113, 60, -71,
            48, -117, 79, -42, -43, 37, -96],
          [13, -52, 48, 6, -8, -55, 35, 75]
        ],
        "output": null
      },
      // ...
    ],
    "attribute": [
      0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0
    ]
  },
  // ...
]
```

Given a corpus of programs, it is possible to automatically generate a *vast* amount of training data for the neural network in the above format. This involves simply taking the program P, placing a constraint on the output value within a certain range, then propagating those constraints backward through P. After this, we can randomly sample inputs from their respective ranges to produce valid examples in E.

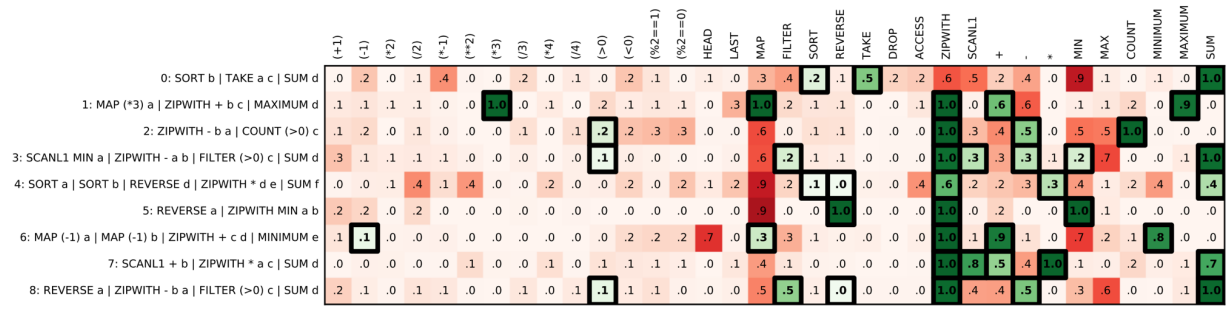


Figure 21: Predicted attributes for various DeepCoder programs; black indicates ground truth.

Exercise 13.1. Obviously, the efficacy of this deep learning model rests squarely on the quality of the sampled data, as the programs must be relatively diverse, and the set of examples must be representative of the program. Does the above algorithm work well, or would you design it differently?

The authors mention that they were inspired by [MTG+13]. However, unlike this older paper, they generate much more training data through their automatic approach, which allows them to train more complex neural network models. As the authors write on page 3 of their paper:

> "The key in the LIPS formulation is to ensure that it is feasible to generate a large dataset (ideally millions of programs)."

Once the deep learning model is trained, you can see some of the output predictions in Fig. 21. Notice that the green marks indicate places where the algorithm did well by correctly predicting the attribute. However, there are also some squares that are deeply red, which indicate that some attributes were mispredicted. The light-green squares are also of major concern. Correct predictions improve the algorithm as a whole, so we will see in the results how this performs.

The details of the neural network architecture are described in the paper. The set E is turned into a fixed-length vector, where each type is one-hot encoded, each vector is padded, and each integer is translated using a 20-dimensional learned embedding. Therefore, integers are limited to the range $[-256, 255]$. You can see a projection of this learned embedding in Fig. 22.

The architecture as a whole is very simple. Each input-output example is passed into a time-distributed fully-connected, feedforward network with three hidden layers. The output logits of the last layer are averaged across all examples (average pooling), and this final vector is passed through a sigmoid activation to obtain the final attribute vector a.

Exercise 13.2. How might this system be extended? You could encode other input-output types by simply embedding them using standard dimensionality reduction techniques, like word embeddings or transformers for strings. Similarly, you might be able to get away with a smarter embedding for arrays, although the authors weren't able to get this to work with GRUs.

Exercise 13.3. The current attribute vector provides a somewhat "impoverished view" of the program, as Joey points out. What if we wanted to extend this to more complex attribute vectors that encoded finer details, such as context-sensitive information? This has the advantage of passing more information to the synthesis system, but it's also likely harder to train.

Finally, I'd like to conclude this section by bringing attention to similar architectures seeing success in the deep learning community, particularly in computer vision.[8] For example, global

[8] This is partly because of my own research experiences at Nvidia.

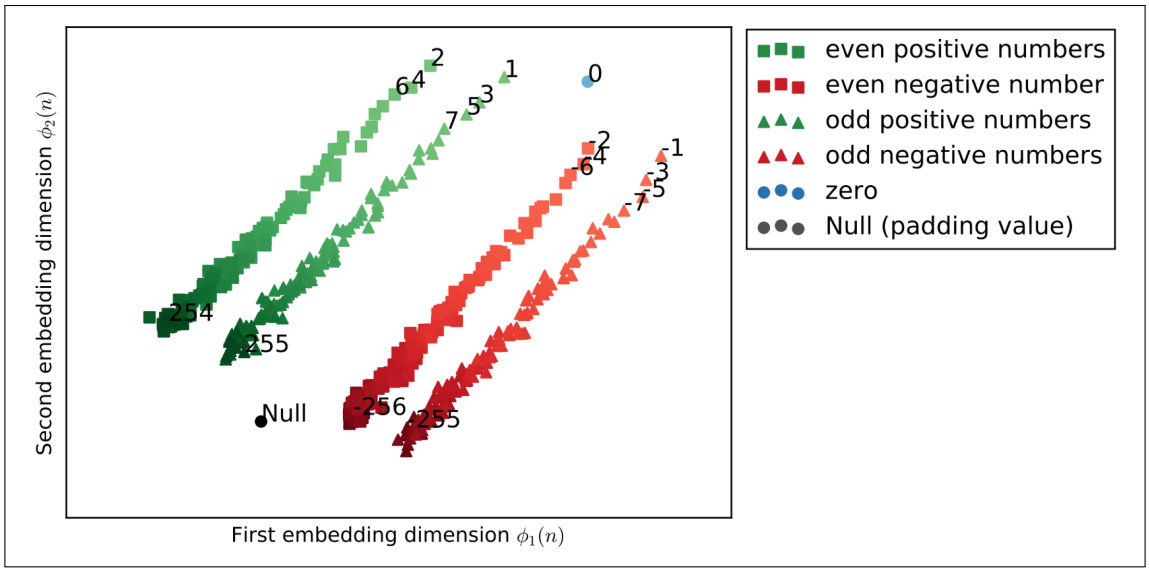


Figure 22: Learned embedding of integers in DeepCoder.

average pooling is a key element in the Atrous Spatial Pyramid Pooling (ASPP) architecture, and variants of this have achieved state-of-the-art results in semantic segmentation for a number of years [CPSA17].

We also might be able to achieve potentially better results by applying *attentional* mechanisms to the output of these parallel feedforward networks. For example, temporally distributed networks [HCW+20] have used attention in a similar manner after a time-distributed network, and hierarchical multi-scale attention [TSC20] has also shown the promise of using attention to combine observations from multiple parallel networks.

13.3 DeepCoder Search Algorithms and Results

Now that we have thoroughly discussed predicting attribute vectors using machine learning, let's see some examples of how this can be applied to actual algorithms. The paper tests out four existing program synthesis techniques, with and without augmentation.

- **Depth-First Search:** Search through all programs of length $\leq T$, with iterative deepening. A solution is found if it agrees with all $M = 5$ examples, and the C++ implementation runs at 3×10^6 programs per second.
- **"Sort and Add":** Similar to depth-first search, but maintaining an active function set, rather than the entire history. This reduces memory requirements, at the expense of potentially revisiting programs multiple times.
- **Sketch:** An SMT-based program synthesis tool, which works by filling in "holes" in a program's source code [SLTB+06].
- **λ^2:** A tool from the programming languages community, which does enumerative search and deduction from a small library of functions [FCD15].

They mention that an RNN-based decoder didn't work for them. I find this somewhat unsurprising, given that they're not backpropagating losses through the entire system. How would it be possible

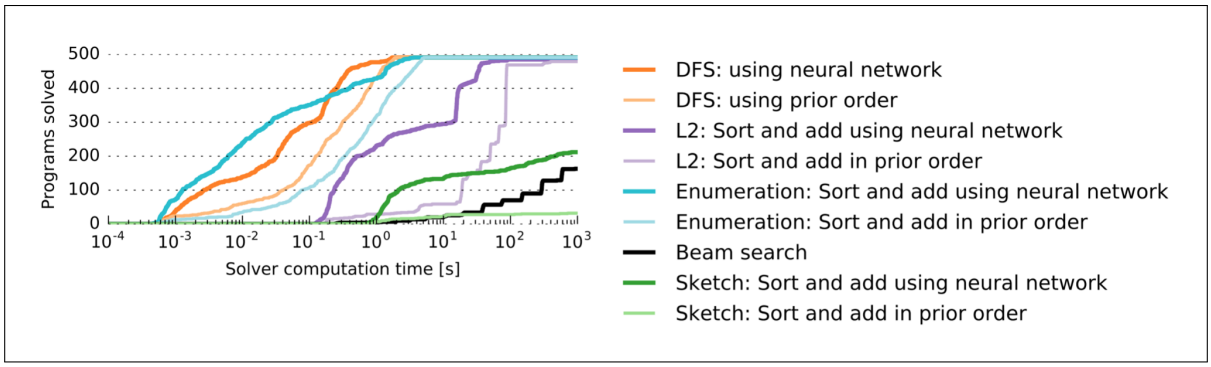
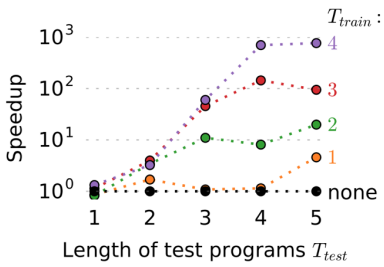


Figure 23: Number of test problems solved versus computation time.

Timeout needed	DFS			Enumeration			λ^2
to solve	20%	40%	60%	20%	40%	60%	20%
Baseline	163s	2887s	6832s	8181s	$>10^4s$	$>10^4s$	463s
DeepCoder	24s	514s	2654s	9s	264s	4640s	48s
Speedup	**6.8×**	**5.6×**	**2.6×**	**907×**	**>37×**	**>2×**	**9.6×**

(a)



(b)

Figure 24: Search speedups on programs of length $T = 5$ and influence of length on speedups.

for an RNN to learn the map $a \mapsto P$, when there can be many programs corresponding to the same set of functions? It seems like a bit of a straw man to compare it like this.

A graph of results is shown in Fig. 23. Observe that the solid lines, which are learning-augmented variants, generally synthesize correct programs much faster than their non-augmented counterparts. This is therefore very promising. The authors also quantify this observation with a direct comparison between computation times, as shown in Fig. 24.

Note. What could be some real-world implications of this synthesis system? Well, in addition to things like *FlashFill*, which has shown how program synthesis can be deployed in applications, a promising area is intelligent code completion. For example, consider the TabNine system, as well as recent research from Microsoft that is being used in Visual Studio Code [SLH+20].

Summary: By learning a self-contained representation of the *attributes* corresponding to a program from input-output examples, DeepCoder (and more broadly, LIPS) is able to use black-box deep learning to augment existing program synthesis algorithms with helpful information that can be used to guide their search.

14 October 20th, 2020

Today, Alex Li will present "Learning programs by learning from failures" [CM20]. This is a very recent paper in the area of inductive logic programming.

14.1 Learning From Failures [CM20]

Recall that *inductive logic programming* is a specific instance of inductive program synthesis, but focused primarily on the case of synthesizing logic programs. These are programs in languages similar to Prolog and Datalog, with Horn clauses. The stated advantages are:

- Requires few training examples.
- Lends well to explainability.
- Naturally supports lifelong and transfer learning.

The problem in general with program synthesis is that search space is huge. For example, two popular current approaches are *example-driven search* and *answer-set programming*, which both try to reduce the search space in various ways. However, the former approach suffers from poor performance on recursive relations, and the latter suffers from poor generalizability.

For some terminology, we call a clause a set of literals with one head, i.e., a Horn clause. A program is simply a set of Horn clauses.

Definition 14.1 (Complete and consistent)**.** We call a logic program *complete*, with respect to a set of examples, if it generates all positive examples. Similarly, we call it *consistent* if it does not generate any of the negative examples in the set.

The *learning from failures* (LFF) approach repeatedly evaluates hypotheses against a set of examples. If the example is not consistent, then it is too general, so we can prune off hypotheses that are a superset of the current one. Otherwise, if the example is not complete, then it is too specific, so we prune off sub-hypotheses.

In order to model this idea, we would like to use the *entailment* partial order, where $P \implies Q$ if the relations generated by P are a subset of Q. Unfortunately, entailment is undecidable, so we use the following proxy.

Definition 14.2 (Theta-subsumption)**.** We say that a clause C_1 *subsumes* clause C_2 if there is a variable substitution θ such that $C_1\theta \subseteq C_2$. In general, If C_1 subsumes C_2, then $C_1 \implies C_2$.

We can also compute subsumption on the level of clauses. For example, consider the following set of three rules.

```
1 C1       = f(A, B) :- head(A, B).
2 C1_theta = f(X, Y) :- head(X, Y).
3 C2       = f(X, Y) :- head(X, Y), odd(Y).
```

You can see that C_1 and C_2 are not directly subsets of each other, but under the renaming operation θ which takes $A \mapsto X$, $B \mapsto Y$, we have that $C_1\theta \subseteq C_2$. Theta-subsumption is a *partial order*, in the sense that it is transitive and has no nontrivial cycles.

Definition 14.3 (Generalization)**.** We call T_1 a *generalization* of T_2 if T_1 θ-subsumes T_2, and we write this as $T_1 \preceq T_2$. Analogously, T_2 is a *specialization* of T_1, and $T_2 \succeq T_1$.

h = { rev(A,B):- head(A,B).
      rev(A,B):- tail(A,C),head(C,B). }

```
included_clause(Cl,id2):-
  head_literal(Cl,rev,2,(V0,V1)),
  body_literal(Cl,head,2,(V0,V1)),
  V0!=V1.
included_clause(Cl,id3):-
  head_literal(Cl,rev,2,(V0,V1)),
  body_literal(Cl,tail,2,(V0,V2)),
  body_literal(Cl,head,2,(V2,V1)),
  V0!=V1,V0!=V2,V1!=V2.
:-
  included_clause(Cl0,id2),
  included_clause(Cl1,id3),
  Cl0!=Cl1,not clause(2).
```

Figure 25: An example of a specialization constraint for a simple hypothesis.

In the example below, we have a subsumption on the level of *theories*, not just clauses. Note that Horn clauses implicitly model disjunctions when relations are duplicated on the left-hand side.

```
% hypothesis h_1
f(A, B) :- head(A, B).

% hypothesis h_2
f(A, B) :- head(A, B), odd(B).

% hypothesis h_3
f(A, B) :- head(A, B).
f(A, B) :- reverse(A, C), head(C, B).
```

In the above example, $h_1 \preceq h_2$, $h_3 \preceq h_1$, and $h_3 \preceq h_2$. Note that $h_3 \preceq h_1$ because h_3 has multiple clauses, which are treated as a disjunction.

14.2 Popper ILP System

Popper is an instantiation of the learning from failures approach that has three steps. During generation, potential hypotheses are created using *answer-set programming* on constraints based on a set of background knowledge (relations), as well as prior knowledge about generalization and specialization. Then, the hypotheses are tested, and additional constraints are added based on whether the hypothesis is consistent or complete. This process can repeat.

As an example, in Fig. 25, you can see a simple example of a program and its corresponding specialization constraint encoded in an ASP solver. This is interesting, and the automated transformation from hypotheses into constraints is the key innovation behind Popper's formulation. The paper also details generalization and elimination constraints, which are similar.

The paper details several experiments illustrating Popper's performance and comparing it to other state-of-the-art ASP systems. In the *Buttons* experiment, various ASP systems are asked to learn a monotone disjunction of n Boolean variables out of 100. The results are shown in Fig. 26.

In other examples, the *Robots* experiment controls an actor on a finite, two-dimensional grid. The logic program is tasked with finding a program that moves the robot to the end of the grid. Interestingly, in this experiment, Popper does *better* when the grid size increases, likely because many non-generalizable hypotheses are pruned.

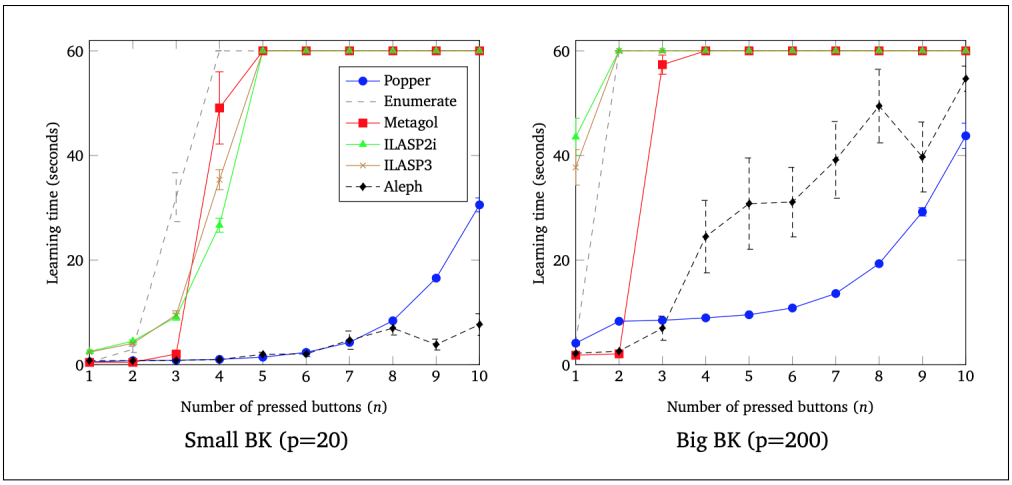


Figure 26: Buttons experiment.

Three more examples were taken, for list transformations, scalability, and sensitivity. The latter two experiments were primarily comparisons with *Metagol* [CT19].[9] They saw that popper's performance improved in all shown axes compared to Metagol.

Note. Given how specific these examples are (and the lack of standardized benchmarks), these results might have been cherry-picked. It's hard to say without taking closer reading.

We have breakout discussions now. I'm speaking to Alex Li, Adam Palay, and He Sun.

- Can we augment the Popper system with something like LIPS, where we guide the search?
- It's interesting that even for simple problems like monotone disjunction, which have tractable polynomial-time solutions (PAC), these inductive logic program synthesis algorithms have trouble. We would expect it to take like 5 ms, not 50 s for the $n = 10$ case. *Can we have program synthesis that works well for simple tasks, while also solving complex tasks?*
- Is the bottleneck in these experiments the answer-set programming part, or the constraint generation part? This would inform which optimizations can be made.
- Can we also apply the learning from failures idea to other paradigms, such as generating functional programming?
- Hypothesis generation is offloaded to the Python-based Clingo ASP system [GKKS14]. How does this system choose when there are multiple possibilities; is it deterministic? It seems like neural network augmentation could be used primarily at this level.

Learning from failures is a very neat paradigm, in the sense that real-life learning with feedback has a lot of parallels, with teachers helping students learn to read/write. This ties into the MAT learning model [Ang87], but we could replace the oracle with a black-box neural net.

[9] Note that this system was developed by mostly the same authors, and it was also published in the *Machine Learning* journal.

15 October 22nd, 2020

Today, Adam Palay will present the paper "Generative Language Modeling for Automated Theorem Proving" [PS20]. Adam mentions that he is a joint student between the engineering and business schools, so he will cold call on people to ask questions about philosophy related to this.[10]

15.1 An Introduction to Theorem Provers

Theorem proving is a general domain where you formalize mathematical theorems into formal languages. Generally, the idea is to combine high-level *tactics* and *substitutions* to repeatedly decompose a goal into subgoals, until you arrive at axioms or definitions.

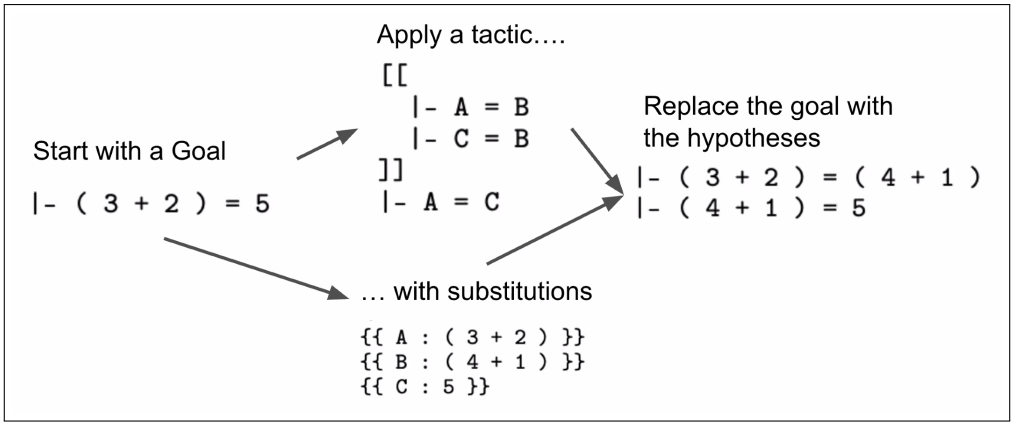


Figure 27: Basic diagram of theorem proving

Examples of theorem provers include Lean and MetaMath (which is used in this paper). These theorem provers are actually used by mathematicians to formalize their logic, so the tactics need to be sufficiently granular to model the kinds of elementary steps written by mathematicians in papers. MetaMath, in particular, goes backward from goal to axioms.

15.2 GPT-f: Automated Theorem Proving [PS20]

The first step of the automated system is to call a learned function f_T : Goal $\to$ [Tactic]. Given any current goal, this learned function generates an array of possible tactics that could be used as the next proof step, along with likelihoods. This is done character-by-character from the goal, using a transformer language model, with training samples of the form:

```
GOAL [[ ]] |- ( ( ( J e. Nrm /\ f e. ( J Homeo K ) ) /\ ( x e. K /\ y e. (
( Clsd ‘ K ) i^i ~P x ) ) ) -> ( ‘’ f " x ) e. J ) PROOFSTEP [[ |- ( ph ->
ps ) |- ( ph -> ch ) |- ( ( ps /\ ch ) -> th ) ]] |- ( ph -> th ) {{ ch :
x e. K }} {{ ph : ( ( J e. Nrm /\ f e. ( J Homeo K ) ) /\ ( x e. K /\ y e.
( ( Clsd ‘ K ) i^i ~P x ) ) ) }} {{ ps : f e. ( J Cn K ) }} {{ th : ( ‘’ f
" x ) e. J }} <|endoftext|>
```

Adam Palay and Felix Sosa then go into a bunch of philosophical discussion (and cognitive science) related to how transformer-based models learn to complete logic from prefix information. This

[10] This is somewhat questionable, but okay.

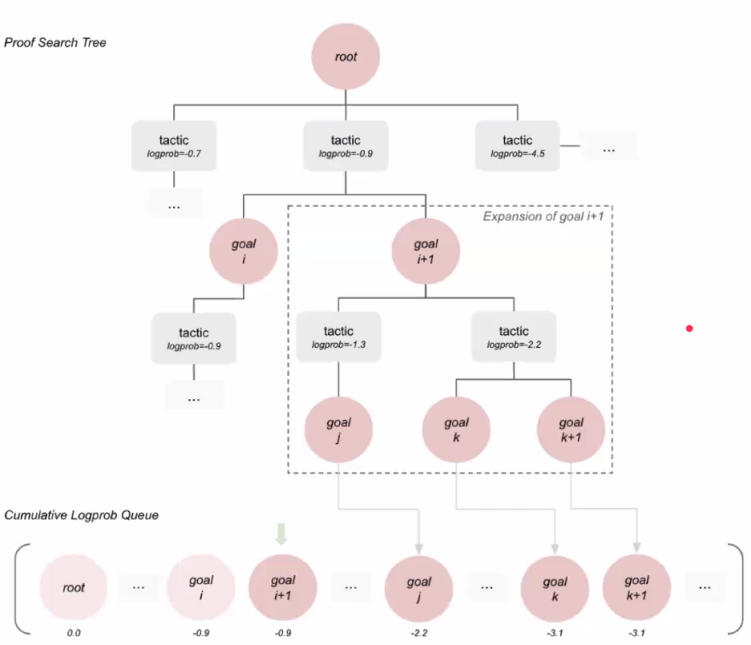


Figure 28: Branching BFS proof tree used for theorem proving.

discussion is somewhat odd, so I won't discuss it more here (but for some reason he ended up analyzing a clip from Disney's *Frozen*).

After training this neural network model for estimating likelihoods of tactics at every step, we proceed by a weighted breadth-first search on the entire proof tree. In each case, we will prioritize goals based on the *cumulative log-probabilities* of all tactics that generated that goal. See Fig. 28 for a visual depiction of the search tree.

This is already a fairly neat and reasonable architecture, but the authors of this paper make one additional innovation. They introduce a learned *provability function* $f_P : \text{Goal} \to [0, 1]$, which outputs the probability that a certain goal can actually be proven. This provability function essentially does a sort of *learned pruning*, where goals that have been visited many times (and cannot be proven) are reweighted to be less important, using the provability function. This essentially adds an *exploration bias* to the exploration vs exploitation landscape.

Note. It's a little bit tricky to train this provability function f_P, as we need to generate negative examples. It turns out that f_P is actually trained *during* proof synthesis as the BFS solver visits unproductive branches.

In the language of the paper, f_T represents a policy function, and f_P represents a value function. Both of these are trained iteratively by analogue of value and policy iteration (see, for example, Markov Decision Process). A comparison of results is shown in Fig. 29, and you can see that adding f_P indeed improves performance marginally. However, we are still generally unable to learn the majority of proofs in the test set.

Note. Another trick used by the paper is the addition of *synthetic data* to their dataset of many real, open proofs in MetaMath. Unlike systems like Lean and Coq, which provide powerful tactics like `ring` and `cc`, MetaMath is much lower-level. Therefore, the paper adds some 1% of synthetic data, simply illustrating n-digit arithmetic and ring algebra to the transformer model. This provides additional supervision for a certain kind of mechanical proof.

Model	Iteration 0	Iteration 1	Iteration 2
160m (*WebMath*) *policy only*	34.79%	38.17%	38.34%
160m (*WebMath*) *policy+value*		39.27%	40.70%
700m (*WebMath*) *policy only*	42.56%	42.23%	43.15%
700m (*WebMath*) *policy+value*		44.59%	**47.21%**

Figure 29: Results of learned policy and value search on theorem proving.

Combining all of the tactics above (including dataset augmentation), along with a very large model with θ_{700m} parameters, and repeated proof attempts, we can do a lot. The final accuracy on the test set was just over 56%, which is not that bad.

Note. Alex Li makes the point that this is extremely similar to systems like AlphaGo [SHM+16] and AlphaStar [VBC+19], which also use a reinforcement learning algorithm based on learned Monte Carlo tree search. The flavor of this paper is similar, showing that deep reinforcement learning can solve *yet another* problem. However, it may not be that inspiring from the theorem-proving view.

The authors solicited feedback from the MetaMath community. Generally, this was positive, mostly focusing on how short the proofs were. However, they don't necessarily indicate anything about the theorem prover generating *new ideas* or more beautiful proofs.

Note. Joey makes the point that oftentimes, shorter programs might be less readable or even less efficient. For example, a short algorithm might take $O(n^3)$ time to run, but by adding a couple more lines, you could optimize it to $O(n)$ time.

16 October 27th, 2020

Today, He Sun will present a lecture on the *AlphaGo*-series of papers, about deep reinforcement learning for playing multiplayer competitive games. This includes *AlphaGo* [SHM+16], *AlphaGo Zero* [SSS+17], *AlphaZero* [SHS+17], and finally *MuZero* [Dee19].

16.1 Reinforcement Learning and AlphaGo [SHM+16]

We start with a brief review of reinforcement learning. Recall that we often model reinforcement learning as a *Markov Decision Process (MDP)* or sometimes a *Partially Observable Markov Decision Process (POMDP)*, where we have a set of states S, actions A, transition probabilities, and rewards $R(s, a)$. Our agent tries to find a *policy* $\pi : S \to A$ that maximizes the total expected reward,

$$\mathbf{E}\left[R \mid \pi\right] = \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid \pi\right],$$

where $\gamma < 1$ is some discount rate that decays the rewards so that this summation converges on the infinite horizon. For an in-depth introduction to MDPs and POMDPs, see [RN02].

Now we introduce AlphaGo. The idea of this algorithm is to apply supervised learning to the policy $p_\sigma(a \mid s)$, which trains a maximum likelihood estimator matching human demonstration:

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a \mid s)}{\partial \sigma}.$$

This creates some demonstrated policy p_σ, which mimics expert human play. The authors then use $p_\sigma(a \mid s)$ to *initialize* the policy p_ρ, which is trained via self-play and reinforcement learning to exceed human performance. In other words, the policy gradient $\Delta\rho$ is given by

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t \mid s_t)}{\partial \rho} z_t,$$

where $z_t = \pm r(S_T)$ is the terminal reward at the end of the game, according to the current player ($\pm$ because zero sum). In addition to the policy network, we also do reinforcement learning for the *value network*, which estimates the value of a state based on what we've seen so far. We obtain the gradient

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial \theta}(z - v_\theta(s)),$$

trained on state-outcome pairs (s, z). The goal here is to minimize the mean-squared error between the predicted value $v_\theta(s)$ and the actual value z, using stochastic gradient descent.

Definition 16.1 (Monte Carlo tree search (MCTS))**.** The *Monte Carlo tree search* algorithm searches through a game tree by randomly exploring nodes based on an *exploration vs exploitation* tradeoff, encoded in any number of heuristics. After each full game in training, the current neural network is trained on all ancestors of the current path. During inference time, the game is partially explored through an iterative Monte Carlo algorithm, but not with full games; the value network is called on leaves of some depth instead. See Fig. 30.

Already, AlphaGo was considered a great breakthrough in deep reinforcement learning, as its results showed that it could beat top human players using a reinforcement learning approach.

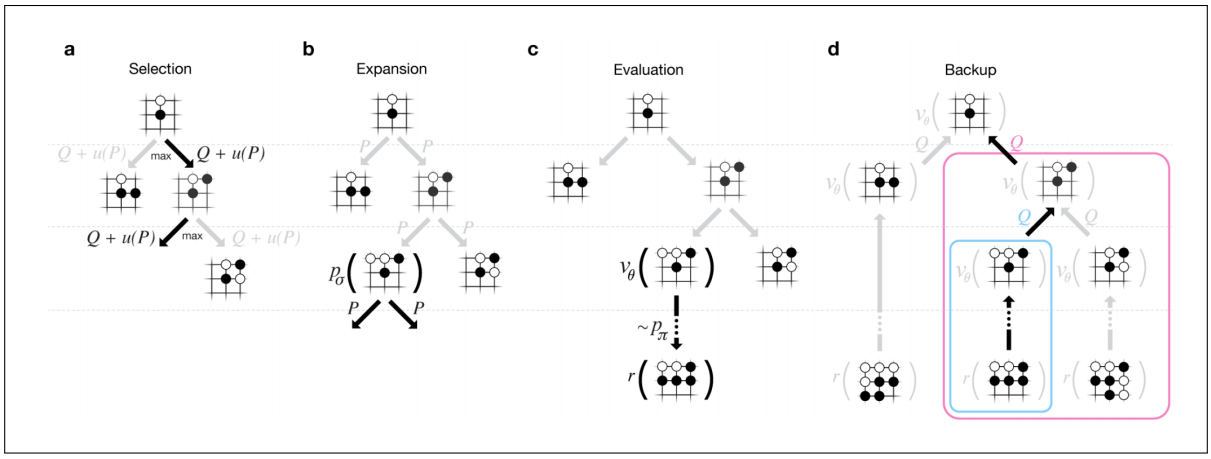


Figure 30: Monte Carlo tree search algorithm in AlphaGo.

16.2 AlphaGo Zero and AlphaZero [SSS+17, SHS+17]

The next paper from the same group, published soon after AlphaGo, was similarly considered a breakthrough. *AlphaGo Zero* differed from AlphaGo in a few ways:

- No human supervision, or prior knowledge of expert strategies.
- Input features are very low-level, just black/white cells.
- Approach is fully based on reinforcement learning and self-play.
- Trains a single, combined policy-value network, instead of separate networks.

A general outline of the algorithm is shown in Fig. 31. To train the model, we use stochastic gradient descent optimizing a certain *combined* loss for both the policy and value,

$$(\mathbf{p}, v) = f_\theta(s), \qquad \ell = (z - v)^2 - \pi^T \log \mathbf{p} + c\|\theta\|^2.$$

Here, c is a hyperparameter (temperature) controlling L2 weight regularization, to prevent overfitting. Thus, the key idea is to simultaneously sum over both mean-squared error loss for the value v, and cross-entropy loss for the policy probabilities $\mathbf{p}$.

Results from the AlphaGo Zero paper were very promising, exceeding human play and the previous AlphaGo hybrid expert-reinforcement learning approach. In particular, even though the agent was able to achieve superhuman performance, it inerestingly enough was *worse* than the supervised learning algorithm at predicting moves by human experts! This suggests that reinforcement learning is able to learn *new strategies* that humans have not discovered yet, rather than just executing human strategies perfectly.

The next paper in this chain is *AlphaZero*. This is similar to AlphaGo Zero, applying the basically same algorithm, except it extends to multiple other games (in this case, Chess and Shogi). Here are the key differences between these games:

- In Go, the positions are homogeneous and rotationally invariant, but Chess and Shogi have very specific position embeddings.
- In Go, the possible actions at every state always look somewhat similar (placing a piece on the board), but Chess and Shogi moves can look very different depending on the current situation (check, castling, etc.).

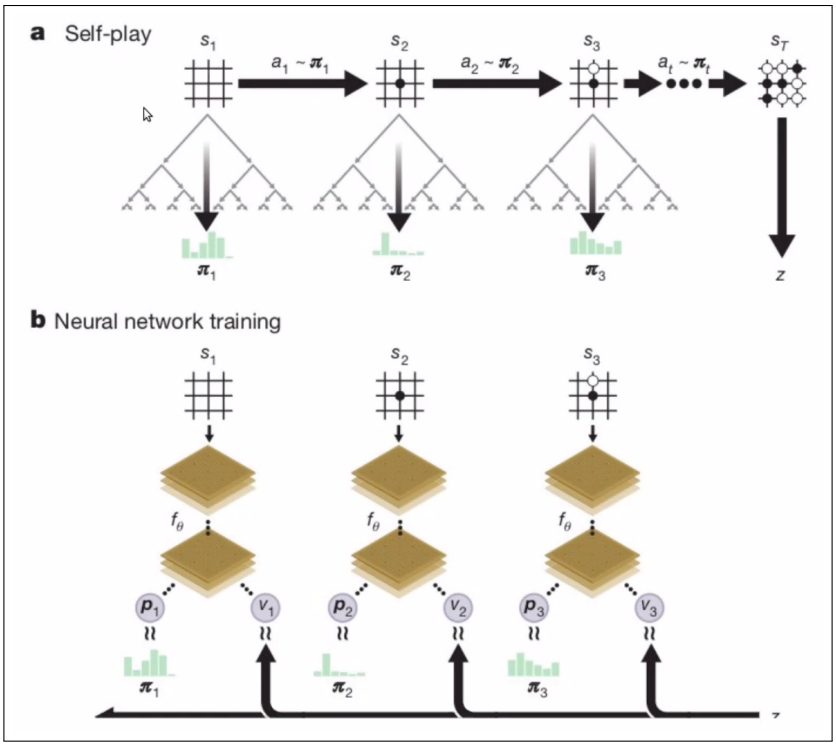


Figure 31: AlphaGo Zero policy search and training algorithm

Besides these differences, which require a slightly different neural network architecture, the results are pretty much the same. After enough steps of training, AlphaZero is able to beat Stockfish (Chess engine) and top Shogi players, as well as AlphaGo Zero, despite being a more general algorithm. However, this may not be too surprising; they just took the same algorithm and just added more compute!

16.3 MuZero: Learning Game Dynamics During Search [Dee19]

The most recent paper in this series is *MuZero*, which significantly generalizes the previous AlphaZero algorithm. There are three key differences:

- While AlphaZero assumes a *perfect simulator*, MuZero only learns the game dynamics (states, actions, and rewards) while playing.
- Conversely, MuZero still assumes a mask of valid actions at each current state, but only at that current state.
- AlphaZero assumes that the game is two-player, while MuZero works for general single-agent POMDP reinforcement learning tasks.

This is a pretty big generalization, as it departs from the assumptions of many traditional algorithms for solving games, where you just simulate a bunch of cases. MuZero predicts three functions using its learned model:

$$\mathbf{p}_t^k, v_t^k, r_t^k = \mu_0(o_1, \dots, o_t, a_{t+1}, \dots, a_{t+k}).$$

This is where the name of the algorithm comes from, as μ_0 is essentially a giant neural network with three heads that predicts policy, value, and rewards in the future. The model is evaluated

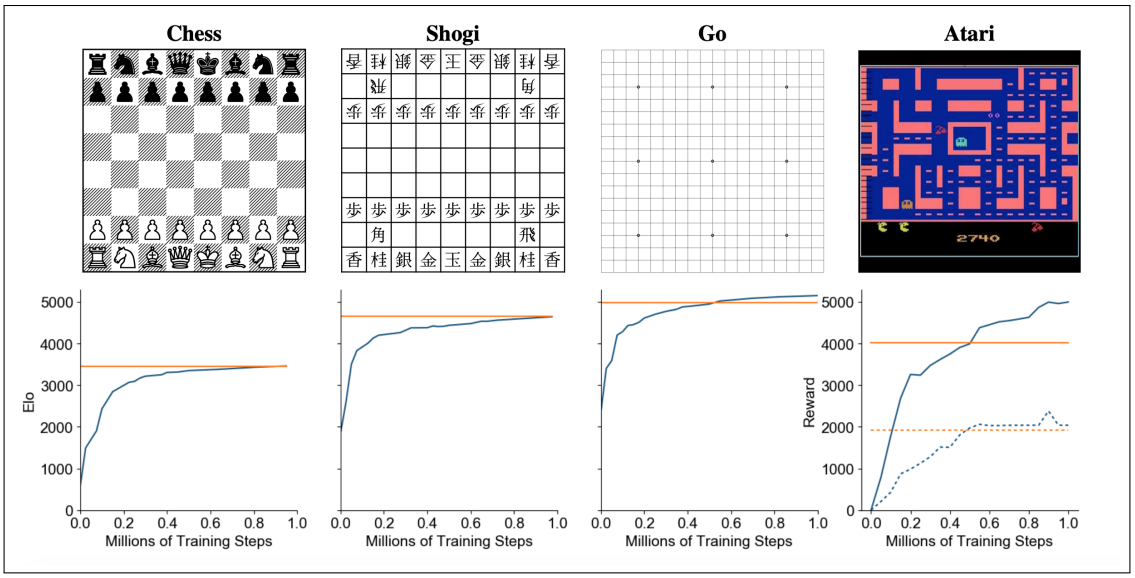


Figure 32: Evaluation examples for μ_0 on various games, showing normalized scores.

on Chess, Shogi, and Go once again, but they change it to not know the rules of the game, and also only provide an *image* of the board as input. The model is also evaluated on an Atari game. Normalized results (Elo scores) are shown in Fig. 32, in particular illustrating that μ_0 matches performance with even top reinforcement learning algorithms.

We had breakout rooms, where we discussed RL and its potential applications to program synthesis. One thing that was mentioned is that *policy gradient methods* using the *score function* reparameterization trick don't fare well on problems with sparse and discontinuous rewards. Quote from another student (Chris Hamblin) who shared his RL experiences:

> I tried to do reinforcement learning at the beginning of my PhD, and I eventually just gave up. Like, I'll never have enough GPUs or compute to actually have these models converge, and none of this will ever produce results.

In some sense, there's a question of whether Google is trying to "mislead" researchers by sharing these cool results that are only obtainable with **massive** amounts of GPU compute. Another student (Felix) mentioned that AlphaGo required $20 million to train, even in its first iteration, therefore requiring a massive logistical support program to execute.

Some ask the question if these kinds of experiments are really pushing the envelope towards generalization,[11] when there's only one massive company can actually train these things. A more charitable interpretation (Joey) is that researchers at DeepMind actually believed that deep RL was the future, and they had the resources to train these large experiments. It's just unfortunate that they were not actually able to make reinforcement learning converge more quickly.

[11] At *CogSci 2018*, Felix mentions that DeepMind really promoted the thesis that humans do not have the ability to fully specify inductive biases to conduct learning for tough problems, and so machines must learn from the ground up whenever possible. Very strong opinion!

17 October 29th, 2020

Today Kevin Ellis will present his work on *DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning* [EWN+20].

17.1 DreamCoder [EWN+20]

Why do we care about program induction? One reason is that programs tend to extrapolate much better to unseen data, and they require less example data to train. This makes them data efficient. Furthermore, they tend to be interpretable, as the primary way humans interact with computers, and they are also Turing complete (universal).

Two of famous examples of program synthesis are FlashFill [Gul11] and Szalinski [NWA+19]. In each example we have a specific DSL that is used to represent certain concepts, and so, one key problem in synthesis is to produce an algorithm that *learns to learn*, by automating the creation and training of such DSLs.

Kevin's solution to this problem, and contribution, is called *DreamCoder*. The idea is that you start with some library of initial primitive functions such as `map`, `fold`, `if`, and `cons`. Then, the algorithm repeatedly generates a learned library of concepts (extending the DSL) at each iteration, which build upon each other to produce progressively more complex concepts. See Fig. 33.

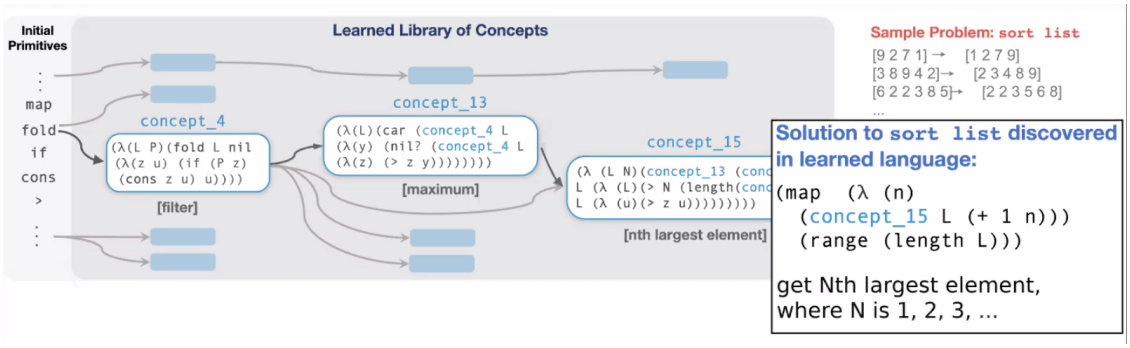


Figure 33: Library of concepts in the process of learning a sorting algorithm.

What's interesting about this kind of idea is that a problem like sorting would be very difficult to understand initially, without a library of concepts. However, once `concept_15` is discovered, the sorting problem becomes much easier to write in just a small amount of code. This motivates DreamCoder as mimicking the way humans solve problems, by combining complex, useful abstractions.

The details of the DreamCoder algorithm draw from neural network literature in Helmholtz machines. These are usually trained using a *wake-sleep* unsupervised learning algorithm, which was a primary motivation of DreamCoder. There are three phases.

1. **Wake:** From our library of input tasks, we use a neural-network guided search to find a soft weighting of the DSL solution space. Then, we search according to this weighting until we either find a solution or timeout. This is similar to [BGB+16].

2. **Sleep:** At this point, we need to spend some time updating two things, our neural network recognition model (for soft weighting), and our library of concepts (abstraction). This is divided into two phases.

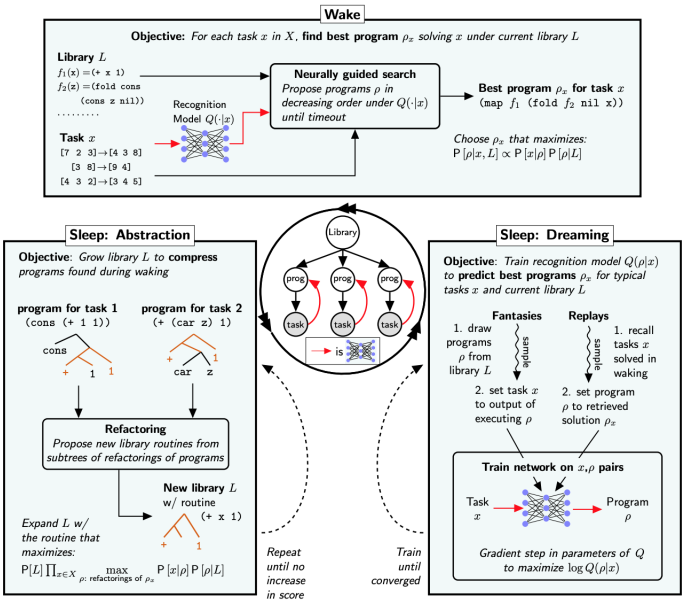


Figure 34: Schematic of DreamCoder's three phases in its wake-sleep algorithm.

(a) **Sleep-Abstraction:** Use a minimum description length algorithm to compress current concepts, and update our DSL to include new concepts.

(b) **Sleep-Dreaming:** Re-train our neural network model by doing experience replay, as well as "dreaming"/sampling new DSL examples, since neural networks require a lot of data to train.[12]

A high-level representation of DreamCoder's phases is shown in Fig. 34. There's an interesting Bayesian interpretation of the compression and dreaming approach. In particular, note that our objective can be written as

$$\Pr(\text{library}) \prod_{\text{task}} \sum_{\text{prog}} \Pr(\text{task} \mid \text{prog}) \cdot \Pr(\text{prog} \mid \text{library})$$
$$\geq \Pr(\text{library}) \prod_{\text{task}} \sum_{\substack{\text{prog} \in \\ \text{beam(task)}}} \Pr(\text{task} \mid \text{prog}) \cdot \Pr(\text{prog} \mid \text{library}).$$

Here, a lower bound holds for any finite beam size (c.f. beam search). The key idea to making this optimization problem tractable is to set a finite bound on the size of the beam. In this case, we will generally set $|\text{beam(task)}| \leq 5$. The first term of this expression, Pr(library), corresponds to wanting to minimize the description length of our library, while the summation corresponds to a learned model sampling programs from the DSL given a task. This is because, by Bayes' rule,

$$\Pr(\text{prog} \mid \text{library}, \text{task}) \propto \Pr(\text{task} \mid \text{prog}) \cdot \Pr(\text{prog} \mid \text{library}).$$

The left-hand side is precisely what the neural network is learning (DSL and task as input), and the right-hand side is in our summation above!

[12] Unlike DeepCoder, which propagates constraints backward to generate input examples, DreamCoder just samples examples uniformly at random from the set of inputs with the correct type. Apparently this was not much of an issue, as the neural network really doesn't have to be perfect. It just needs to provide some slight guidance to the search algorithm.

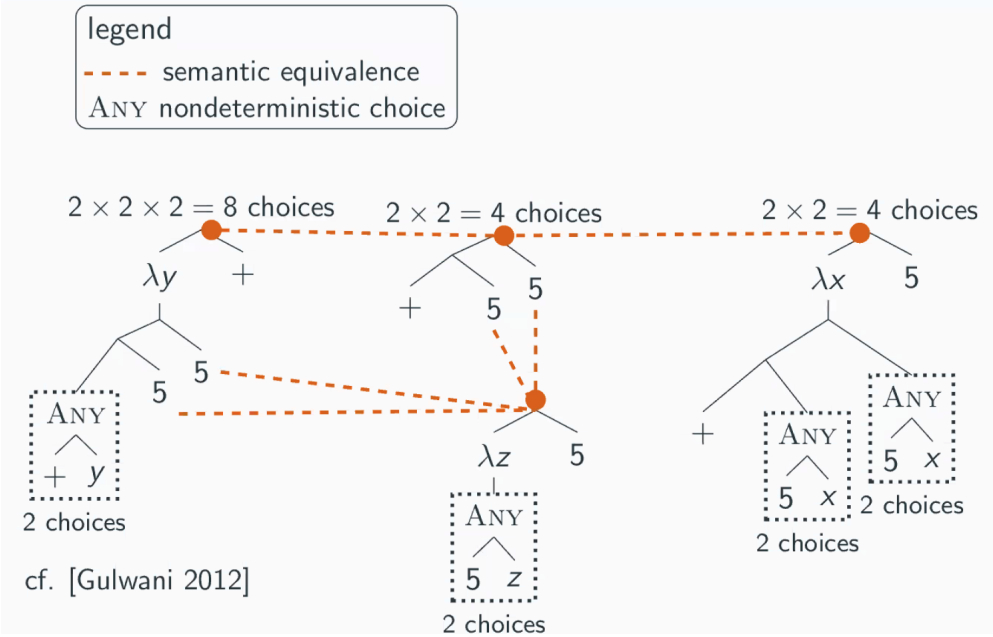


Figure 35: Growing the library of concepts through refactoring.

Note. Mark and Kevin note that this algorithm is very expectation-maximization like. It's almost directly analogous to a particle variational method (e.g., sequential Monte Carlo), where the beam search falls out from the particle sampling step.

17.2 Sleep-Abstraction and Concept Discovery

Now let's see how the Sleep-Abstraction phase works (c.f., *EC algorithm*, from Section 11). We want to grow our library of concepts by building new abstractions, simplifying and compressing our current library to minimize Pr(library). The refactoring algorithm is based on a data structure called an *equivalence graph*, used in compilers for super-optimization. It's also been used in FlashFill [Gul11]. An example of compressing abstractions as "refactoring" programs is shown in Fig. 35.

Note that the refactoring algorithm is polynomial in the size of the program, but exponential in the *amount of refactoring* that you do. This is quite scary, but we can limit the amount of refactoring and use some fancy data structures to make it tractable.

Note. Since we add the Y-combinator to the grammar, we get easy recursion. However, we now have a problem where programs could possibly run for a long time, or even forever. This is hard, so DreamCoder just adds a timeout to this search algorithm. There is a lot of interesting literature on the problem of synthesizing programs that run for a long time, but not much success.

On the other side of the argument, Joey makes the point that some PL researchers are claiming that no reasonable program *requires* being written in a Turing-complete language.[13] In this case, if you just allow things like `fold` and `map`, you can write most programs that you care about. This allows you to introduce type systems in a way that avoids problems with unrestricted recursion. DreamCoder simply uses a Hindley-Milner algorithm. Philosophically though, people still program in Turing-complete languages such as Python in practice. Joey alludes to a somewhat vague argument that people add *implicit* type systems to their code to reason about it.

[13] Edwin Brady has a hot take about this, a challenge to come up with an example that requires Turing completeness.

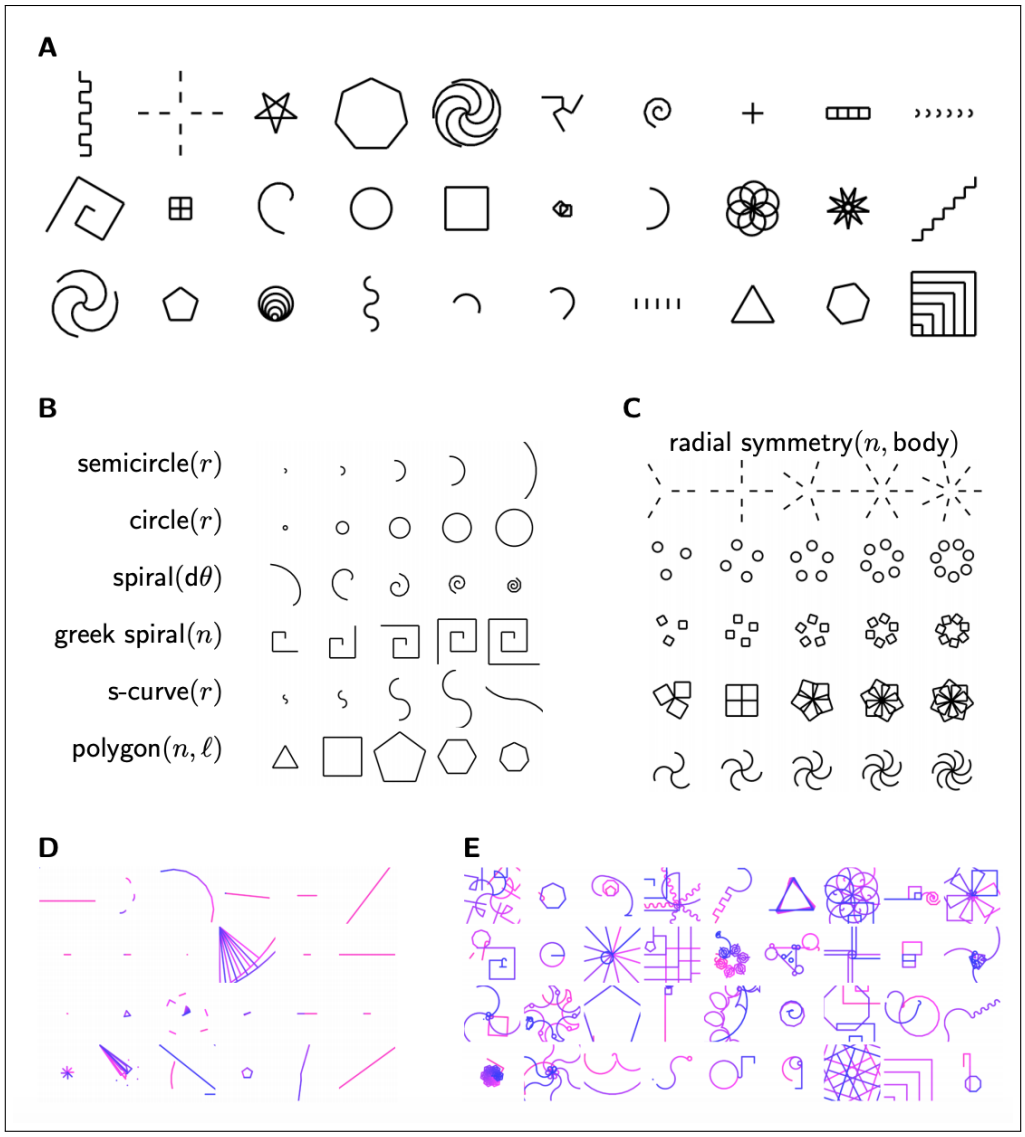


Figure 36: DreamCoder run on LOGO graphics tasks: library routines and dreamed output.

17.3 Neural Network Recognition Model

For the neural network recognition model, we will model program generation as top-down synthesis in the DSL parse tree. In this case, the neural network will be run only once per task to predict the discrete distribution

$$\Pr(\text{child} \mid \text{parent}, \text{task}),$$

which is a vector of probabilities. The motivation here is similar to DeepCoder, but slightly more complex since we also condition on the parent. This allows us to also save on GPU compute cost (as there's basically $O(1)$ neural network inference), so we're bottlenecked by CPU here, not GPU. This allows us to easily write highly parallel program search algorithms.

An example of DreamCoder being evaluated is shown in Fig. 36. Note the nice pictures illustrate the point of *synergistic bootstrapping* in the algorithm, which is very exciting. For example, the library of concepts shown in **B** is very interpretable, and it is even able to learn human concepts

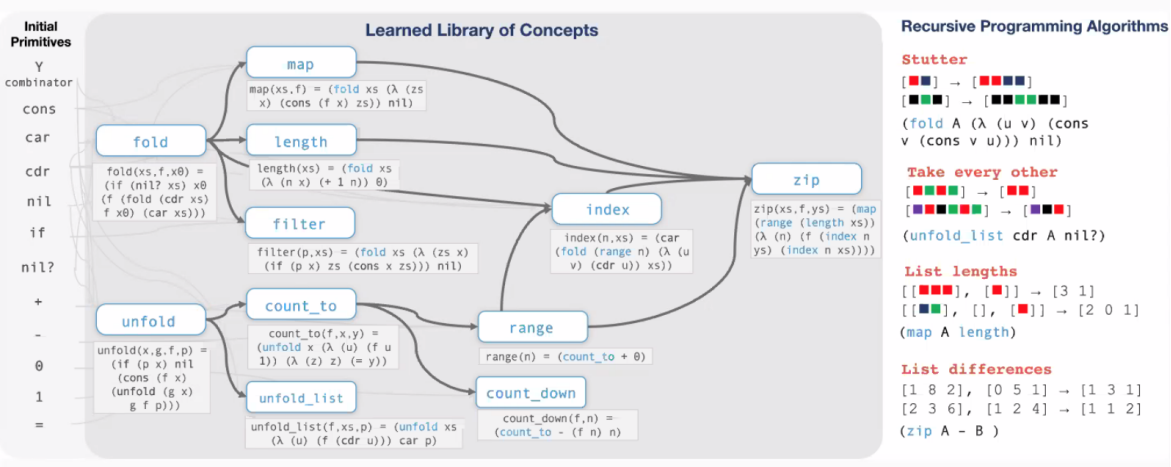


Figure 37: Learning a library of functional programming concepts.

like polygons and box spirals.

Another example of DreamCoder evaluation was in discovering physics and math equations. As Kevin says in his presentation, "We gave DreamCoder a bunch of equations from *AP Physics* and *MCAT* cheat sheets." DreamCoder is indeed able to learn many of these equations, written in a Lisp-like expression grammar, simply from input-output examples. This is really interesting, as we essentially learned neural-guided search over a giant space of expressions, with modular concepts!

One more really interesting example was in learning a library of functions to solve simple functional programming tasks, in lambda calculus. This is shown in Fig. 37. What's interesting is that although we started with the Y-combinator, DreamCoder immediately started by learning concepts for `fold` and `unfold` (named in the PL literature as *origami programming*), and none of the final tasks were actually solved directly with the Y-combinator. This illustrates that even with a minimal basis, we can solve very many problems, but starting from a minimal basis requires either a lot of compute (in this case, about 5 days on 64 cores), or a lot of training data.

17.4 Unresolved Questions

Here's some interesting current problems that people are tackling.

- Not everything is crisp and symbolic. Problems have perceptual parts too (c.f., [MDK+18]). How do we learn DSLs for neuro-symbolic hybrid programs? DreamCoder is neural and symbolic in some sense, but its output search space is still purely symbolic.
- Search is still a very hard problem. See the recent "REPL" paper published by Kevin in NeurIPS 2019 for more about this [ENP+19].
- What if you are given just a library of functions, but no supervision in example tasks to solve? This prevents modular concept discovery algorithms like EC and DreamCoder from working very effectively. Previous work has shown that removing "easy" tasks from the set precludes learning of "hard" tasks. Kevin thinks that learning from scratch with just a library (and no examples) is intractable.

18 November 3rd, 2020

Today we had project pitches from all students. The presentation was published here.

19 November 5th, 2020

Today, Shashank Srikant will present the article "Semantic Robustness of Models of Source Code" [RHW+20].

19.1 Machine Learning Models of Source Code

We begin by asking the question: What are some applications of machine learning in practical software engineering and programming languages work? Examples include TabNine, Visual Studio IntelliCode, and research on automatic bug-finding. The first product that "started it all" was Martin Vechev's tool *JS Nice*, which performs machine-learning augmented analysis of JavaScript programs using nontrivial statistical methods.

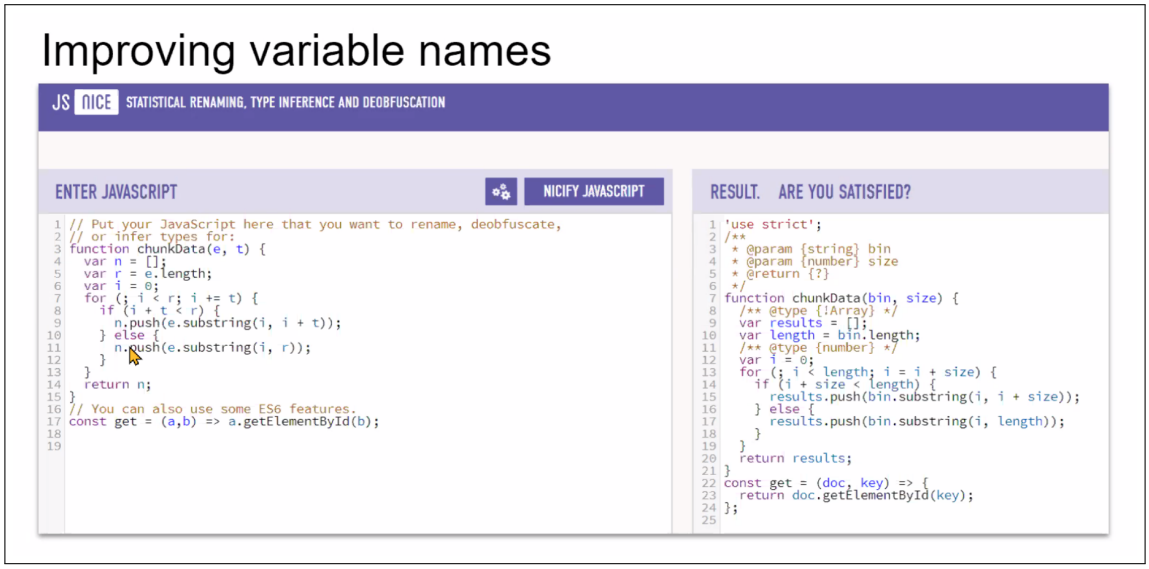


Figure 38: JS Nice tool, browser interface for renaming, type inference, and deobfuscation.

With all of these interesting examples of applications, we should ask how these machine learning models might go wrong. For example, we might have a program synthesis or rewriting model introduce bugs in the source code. Also, in the case of synthesis tools like TabNine, researchers have shown that adversarial examples can be constructed so that the recommender suggests *insecure* modes of encryption [SSTS20]. This is what motivates today's lectuer topic.

19.2 Adversarial Robustness of Source Code Models [RHW+20]

The classic example of adversarial poisoning is in image classification models. There's classic literature on using gradient-based methods to fool convolutional neural networks by adding a very small, imperceptible L_∞ perturbation to the pixels [GSS14]. However, it's less clear how this extends to programming languages — what constitutes as an "similar" program?

Essentially, the idea is to add *obfuscation* to the source code in a way that adversarially fools white-box machine learning models. These are certain, identity transformations that do not change the semantics of the program. If you have enough dimension in your obfuscations, you can probably generate quite bad adversarial examples. Some examples are shown below:

1. `AddDeadCode` – add a useless if statement that does nothing.

(c) Results for the targeted 1-adversary ($\mathcal{Q}^1_{\mathrm{G}}$).

Model	Training	c2s/java-small		csn/java		csn/python		sri/py150	
		Nor	$\mathcal{Q}^1_{\mathrm{G}}$	Nor	$\mathcal{Q}^1_{\mathrm{G}}$	Nor	$\mathcal{Q}^1_{\mathrm{G}}$	Nor	$\mathcal{Q}^1_{\mathrm{G}}$
seq2seq	Tr-Nor	37.8	23.3	32.3	17.2	28.9	16.2	34.3	22.0
	Tr-Aug	38.8	27.8 [+4.4]	32.6	21.4 [+4.3]	29.8	20.9 [+4.7]	33.7	24.0 [+2.0]
	Tr-$\mathcal{Q}^1_{\mathrm{R}}$	40.0	27.5 [+4.2]	38.1	30.1 [+12.9]	36.6	29.8 [+13.5]	41.8	33.4 [+11.4]
	Tr-$\mathcal{Q}^1_{\mathrm{G}}$	40.7	**32.0 [+8.7]**	37.8	**32.8 [+15.6]**	36.2	**32.2 [+16.0]**	41.8	**37.1 [+15.1]**
code2seq	Tr-Nor	41.4	24.6	39.2	19.6	37.4	21.0	39.7	23.7
	Tr-Aug	42.3	23.4 [-1.2]	39.4	19.7 [+0.1]	36.8	20.8 [-0.2]	39.7	24.4 [+0.7]
	Tr-$\mathcal{Q}^1_{\mathrm{R}}$	42.3	28.1 [+3.5]	40.0	23.5 [+3.9]	37.6	22.5 [+1.6]	40.0	26.6 [+2.9]
	Tr-$\mathcal{Q}^1_{\mathrm{G}}$	43.6	**31.6 [+7.0]**	40.6	**27.5 [+7.9]**	37.3	**23.8 [+2.8]**	40.2	**29.5 [+5.8]**

(d) Results for the targeted 5-adversary ($\mathcal{Q}^5_{\mathrm{G}}$).

Model	Training	c2s/java-small		csn/java		csn/python		sri/py150	
		Nor	$\mathcal{Q}^5_{\mathrm{G}}$	Nor	$\mathcal{Q}^5_{\mathrm{G}}$	Nor	$\mathcal{Q}^5_{\mathrm{G}}$	Nor	$\mathcal{Q}^5_{\mathrm{G}}$
seq2seq	Tr-Nor	37.8	11.5	32.3	9.8	28.9	10.4	34.3	16.5
	Tr-Aug	38.8	23.5 [+12.0]	32.6	15.6 [+5.8]	29.8	14.4 [+4.0]	33.7	19.1 [+2.6]
	Tr-$\mathcal{Q}^1_{\mathrm{R}}$	40.0	21.7 [+10.2]	38.1	25.1 [+15.3]	36.6	25.3 [+14.9]	41.8	28.1 [+11.6]
	Tr-$\mathcal{Q}^1_{\mathrm{G}}$	40.7	**26.3 [+14.8]**	37.8	**29.7 [+19.9]**	36.2	**29.4 [+19.0]**	41.8	**33.4 [+16.8]**
code2seq	Tr-Nor	41.4	15.5	39.2	12.1	37.4	15.6	39.7	18.2
	Tr-Aug	42.3	14.5 [-0.9]	39.4	11.8 [-0.3]	36.8	15.8 [+0.2]	39.7	17.8 [-0.4]
	Tr-$\mathcal{Q}^1_{\mathrm{R}}$	42.3	22.4 [+7.0]	40.0	17.6 [+5.5]	37.6	17.6 [+2.0]	40.0	22.1 [+3.9]
	Tr-$\mathcal{Q}^1_{\mathrm{G}}$	43.6	**27.3 [+11.9]**	40.6	**22.0 [+10.0]**	37.3	**20.6 [+5.0]**	40.2	**25.1 [+6.9]**

Figure 39: Results of an adversarial synthesis attack.

2. `RenameLocalVariables` – replace a random local variable with a hole.
3. `RenameParameters` – replace a random parameter's name with a hole.
4. `RenameFields` – replace a random field with a hole.
5. `ReplaceTrueFalse` – replace a random Boolean literal with a hole, such as `x == x`.
6. `UnrollWhiles` – unroll the loop body of a random while loop by one iteration.
7. `WrapTryCatch` – add a useless try/catch error handling statement.
8. `InsertPrintStatements` – insert a random print statement to the program.

Note that the idea of a *hole* used above can be replaced with any token literal that does not conflict with the rest of the program, which gives us a lot of flexibility. This terminology comes from Armando Solar-Lezama's work on the *Sketch* program synthesis tool [SLTB+06].

```
def foo():
    x = foo1()
    y = foo2()
    print("Hello World")
```

As a case study, let's say that you have the simple Python program given above. One thing that we could do is replace the local variables with holes, and another possible maneuver is to rename the functions arbitrarily. However, as the edit distance increases, we get a combinatorial explosion in the number of possible transformations. Finding adversarial examples then becomes an optimization problem.

The approach taken by our discussed paper is to instead model a k-adversary, which is allowed to make up to k transformations on the program. Because this problem is kind of difficult, they are not necessarily able to direct their search well, so they're just going to randomly sample 10 sequences of transformations and take the average response. They evaluate for $k = 1$ and $k = 5$.

The interesting part of this paper is that they use *gradient descent over the vocabulary of tokens*. In other words, at each hole in the program created by a transformation, their synthesis tool tries to do gradient ascent on $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$ to find an adversarial token replacement. The rest of the paper, besides this simple idea, is just padding with a bunch of math.

Results of the tool are shown in Fig. 39, in the particular case of targeted attacks. It's clear that a small, simple change in the program can fairly significantly decrease the accuracy. However, for what it's worth, the accuracies are pretty low in any case, and even a random black-box adversarial model (without using gradient descent) decreases the accuracy significantly as a baseline. In image classifiers, adversarial examples are interesting because they take a 99%-accuracy model and completely fool it, but the program synthesis models are not very high-accuracy anyway.

20 November 10th, 2020

Today, Mark Goldstein will present the paper "Predicate Exchange: Inference with Declarative Knowledge" [TBM+19]. Mark mentions that this paper may have some confusing language (as some students mentioned), as it lies somewhere between programming languages and statistics, but this is simply a consequence of the interdisciplinary field.

20.1 Predicate Exchange [TBM+19]

The problem addressed in this paper is roughly of Bayesian inference on *predicates*. Here's a single sentence quoted from the paper that crystallizes this idea.

> Probability theory treats conditioning on predicates and concrete observations uniformly, but sampling from models conditioned on most predicates is challenging due to the lack of a tractable likelihood function.

The idea is that we condition on an event (predicate) that might be very discrete or have a complex topology, rather than a simple continuous random variable. This generalizes Bayesian inference to larger domains that have a complex, possibly high-dimensional constraint.

Proposition 20.1. *Given a joint distribution, conditioning is in general an intractable operation.*

Proof. Consider a set of simple random variables, which may just be i.i.d. Bernoullis. Then, we can construct a very complex one-way function and run it on those random variables, such as a hash function. Assuming that this function is intractable to invert, we cannot condition on the resulting value without, equivalently, inverting the hash function. □

Hopefully this explains why conditional inference is sometimes called *inversion*. From a computer science perspective, you can think of $p(x \mid z)$ as fixing the result of z, which is some function $f(x)$. For simple conditions, we can use MCMC algorithms like Metropolis-Hastings to sample from the conditional distribution using a Markov chain, without needing to explicitly know the normalizing factor. However, complex conditions create difficult topology.

The Metropolis-Hastings algorithm and other MCMC algorithms are very *local*; they have trouble with distributions supported on two disjoint sets. The ability to remain local is what makes their Markov chain mixing time reasonable, for common distributions, but they can't jump between rigid predicates supported on different parts of the space. This is why we need a softer predicate, which is not just $\{0, 1\}$.

Definition 20.2 (Relaxation)**.** A *relaxation* of a predicate $\ell : \mathcal{X} \to \{0, 1\}$ is a function $\tilde{\ell} : \mathcal{X} \to [0, 1]$, parameterized by a temperature $\alpha \in [0, \infty)$, with the following properties:

- $\lim_{\alpha \to 0} \tilde{\ell}(\mathbf{x}; \alpha) = \ell(\mathbf{x})$.
- $\lim_{\alpha \to \infty} \tilde{\ell}(\mathbf{x}; \alpha) = 1$.
- For all $\alpha < \infty$, $\tilde{\ell}(\mathbf{x}; \alpha) = 1$ if and only if $\ell(\mathbf{x}) = 1$.

To make a realization of this relaxation idea, we need to build some basic predicates from the ground up, then combine them using fuzzy logic and various conjunctions. For example, soft conjunction and disjunction, labeled $\tilde{\wedge}$ and $\tilde{\vee}$, simply take the min and max of two soft predicates. However, we run into an issue as soon as we get to negation, as typical soft negation strategies are *one-sided* and violate one or more of our relaxation conditions.

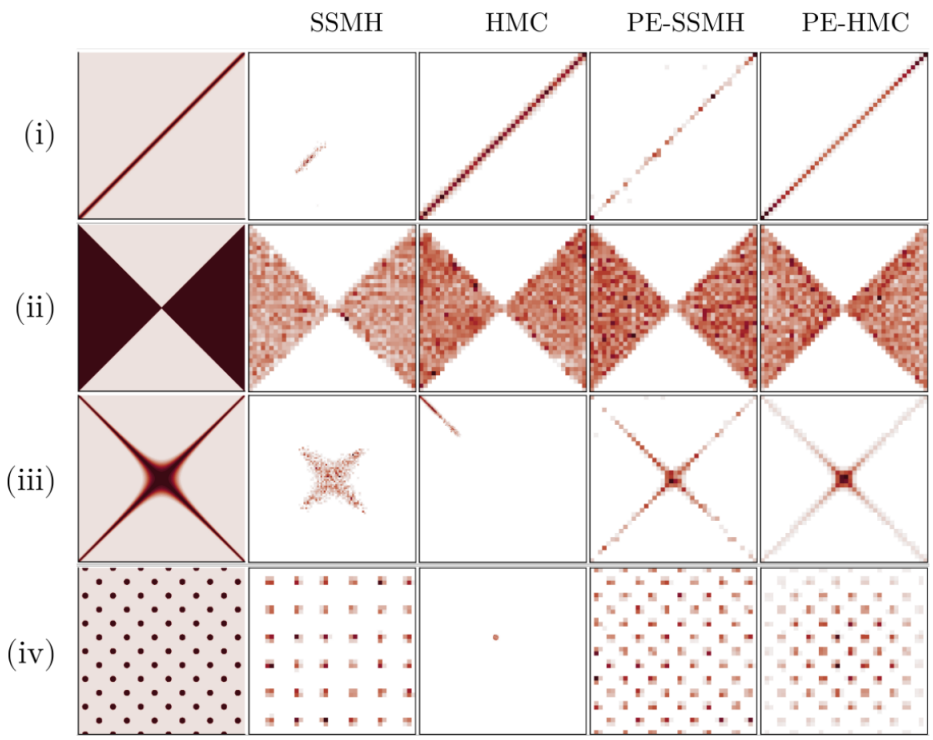


Figure 40: Results of predicate exchange with Metropolis-Hastings and Hamiltonian Monte Carlo.

Instead, two-sided predicates are defined as *pairs* of soft primitives (a_0, a_1), where a_0 is consistent with ℓ, while a_1 is consistent with $\neg\ell$. This allows us to implement negation in a very funny way, $\tilde{\neg}(a_0, a_1) = (a_1, a_0)$, while conjunction and disjunction only have to be slightly adjusted.

Definition 20.3 (Replica exchange)**.** The *replica exchange* algorithm simulates several parallel MCMC trajectories, i.e., copies of Metropolis-Hastings, for chains at varying values of the temperature parameter α. Then, at several points, states from higher-temperature chains are swapped with states from lower-temperature chains. Samples are then taken from the exact chain (which conditions on the predicate being strictly true).

This converges to an unbiased estimator of the actual distribution, since we condition on the predicate being actually true. Thanks to condition 3 of the relaxation, predicate exchange yields a true unbiased estimator, despite adding heuristics to improve convergence speed and avoid getting stuck in local sets.

20.2 Parallel Tempering Results

Results of the predicate exchange algorithm are shown in Fig. 40, for various sets of points in a unit square. We can see that with events supported on small regions, or disjoint sets, the predicate exchange variants perform much better than both the simple Metropolis-Hastings and Hamiltonian MCMC variants. Recent work on extending these results includes [vKAvH20], which compares different fuzzy logic operators (e.g., $a\tilde{\wedge}b = ab$).

Another paper of interest is [ED05], which discusses the parallel tempering algorithm in great detail, with applications to chemistry. This is related to spin-glasses.

21 November 12th, 2020

Today Yafah Edelman will present the paper "Preventing undesirable behavior of intelligent machines" [TdSB+19]. This was published in *Nature*.

21.1 "Seldonian" Machine Learning [TdSB+19]

Before we can talk about preventing undesirable behavior, we first need to define what undesirable behavior actually is. An example provided by the paper is the prediction error (mean error, not mean squared error) of an algorithm on students' academic performance at a school. One tested regression algorithm tended to overestimate men and underestimate women (why?).

Definition 21.1 (Seldonian optimization[14])**.** In traditional computational learning, we attempt to optimize some loss objective f given some hypothesis θ, which can be written as

$$\operatorname*{argmax}_{\theta \in \Theta} f(\theta).$$

In the *Seldonian optimiation problem*, we add several measures $\{g_1, \dots, g_n\}$, which are functions $\Theta \to \mathbb{R}$ measuring *desirability* of an output solution. We also define $\mathcal{A}$ to be the set of feasible algorithms $a : \mathcal{D} \to \Theta$, where $\mathcal{D}$ is the set of possible input datasets, and we change our objective to be $f : \mathcal{A} \to \mathbb{R}$. Then our problem becomes

$$\operatorname*{argmax}_{a \in \mathcal{A}} f(a)$$
$$\text{s.t. } \forall i \in \{1, \dots, n\}, \Pr\Big(g_i(a(D)) \le 0\Big) \ge 1 - \delta_i,$$

where $\delta_i > 0$ are a set of minimum allowable slack parameters for fairness.

Note. One interesting approach is that the paper treats the entire algorithm as a *black box*. This means that our desirable fairness measures g_i are purely statistical, and cannot make assumptions about the actual inner workings of the algorithm.

We now have breakout rooms. I'm talking to Yafah Edelman, Mark Goldstein, and Alex Li. Here are a couple points that were brought up:

- The measure g_i here seems like a *negative fairness* definition. In particular, you need to know which cases of possible unfairness to look for, before you can actually run this algorithm. Every discrimination case must be formalized and explicitly encoded in a constraint g_i, and parameters must be tuned, before a good output can be obtained.
- Since the model is a black box, there's a question of whether an outcome-based definition of fairness necessarily makes any guarantees about whether the internal mechanics are fair themselves. Andrew brings up this point in a discussion post. What if the algorithm somehow adjusted on gender in such a way as to meet the first hard constraint, without actually making its algorithm more "fair" on the inside?
- To be fair to the paper though, they don't claim to be *embedding morality* into the model; they just try to give the end user more control in controlling the model.

[14] They arbitrarily named this concept after a fictional character, Hari Seldon.

- Another weird point about the paper is that their ML classifier is trained on *mean squared error*, yet their fairness measure is based on the average linear deviation. Yafah mentions that the observed fairness problems seem to be an artifact of different outliers in the male and female parts of the dataset. Would this bias still occur if the classifier was trained on least absolute deviation, L_1 error instead?
- Another problem is that the paper claims to target people who are domain experts, yet not an expert in ML or statistics. If ML experts have trouble defining fairness, how can we expect a non-technical person to design appropriate functions g_i?

If you've observed a bias in past models, then you can treat that as a constraint. However, it's hard otherwise, as a product manager or other non-technical person, to formalize biases before actually starting to train the model.

21.2 Optimizing with Fairness Constraints

We spent a lot more time having open comments about the ethics of fairness, and the students from the business school really robustly chimed in on this. Anyway, it's time to start talking about what Seldonian algorithms actually look like.

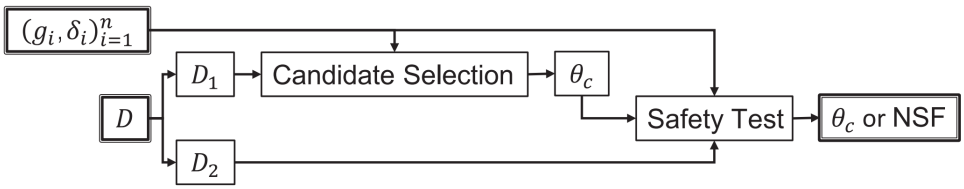


Figure 41: Seldonian regression, which outputs a solution θ_c or "No Solution Found" (NSF).

The basic schematic of a Seldonian algorithm is shown in Fig. 41. It essentially formalizes the concept of using a validation set to test for desirable properties. Additionally, a "Safety Test" is used to ensure that the algorithm has specified fairness properties (this takes the place of g_i in Definition 21.1 above), using either Hoeffding's inequality or hypothesis t-testing.

We also call an algorithm *quasi-Seldonian* if its safety test relies on reasonable but not perfectly rigorous statistical assumptions, such as appeals to the central limit theorem as an approximation. This gives the framework a bit more flexibility.

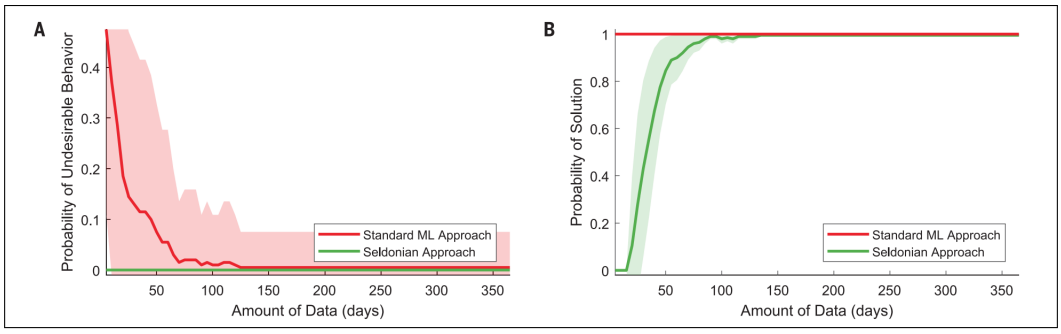


Figure 42: Seldonian reinforcement learning algorithm for bolus calculation in type 1 diabetes.

Finally, Fig. 42 shows some results on a proof-of-principle problem involving blood sugar calculations. As the amount of data increases, the Seldonian algorithm is more easily able to guarantee

that a fair solution has been found, while both approaches converge to a low probability of undesriable behavior. So it seems that the paper's innovation is especially helpful in the low-data regime, which makes sense. Low data is the setting where ML struggles the most!

22 November 17th, 2020

Today, Felix Sosa will present the paper "Program Synthesis from Polymorphic Refinement Types" [PKSL16]. This paper introduces the *Synquid* synthesis tool, which is based on refinement types.

22.1 Polymorphic Refinement Types

Felix mentions that his background is not in programming languages, and while he thought he knew what a type was before reading this paper, he realized that he was not familiar with some of the advanced topics in this paper (refinement types, verification, etc.). Therefore, he provides some type theory background, from various PL courses (Nada Amin, Armando Solar-Lezama).

Generally, *types* are anything that restricts the possible set of possibilities that any value could take along the course of program execution. These can be used in any number of ways to prevent undesirable behavior, catch bugs before they appear, or aid in spec-driven development.

How can types be applied to a program synthesis perspective? Well, one of the main issues with top-down search is a combinatorial explosion in the amount of possibilities to check. Consider the example shown in Fig. 43, which illustrates the typing rules of a simple language. In general, languages all have typing rules expressible in formal logic in this form, although they might get arbitrarily complex (see: Hindley-Milner). The source for the figure is 6.887.

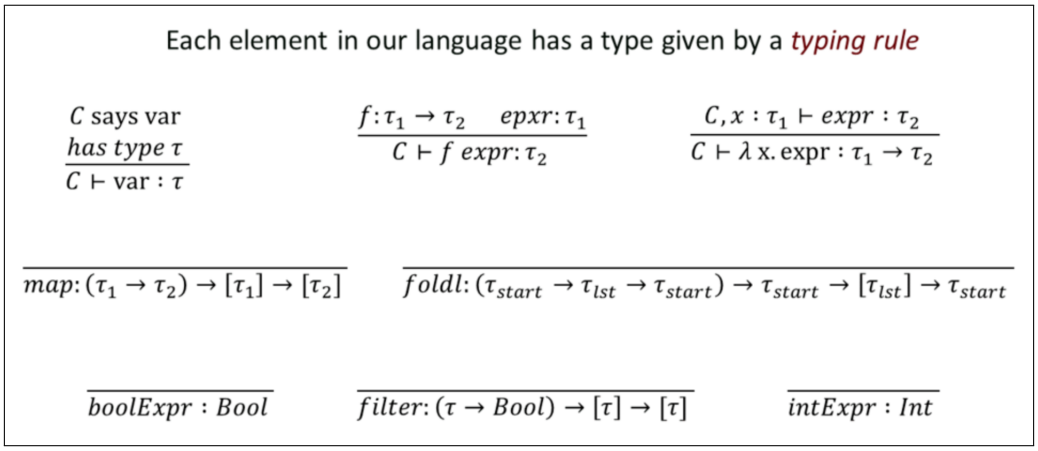


Figure 43: Typing rules for a simple functional language.

We had breakout rooms, where I talked with Mark, Romil, and Chris about type theory foundations. It seems like a good working definition of *type* is just anything that you can associate with a value, based on a formal set of logic rules. One thing I also noticed is that liquid type-based pruning approaches are kind of dual to the technique of *symbolic execution*, where you also use logic solvers, but to provide constraints on the input examples rather than the program itself. Some references for symbolic execution are KLEE, Armando Solar-Lezama, and Emina Torlak.

Now we can finally introduce *liquid types*, or refinement types. The blog post "A Gentle Introduction to Liquid Types" is a good example of this. We also add a certain sub-language to the type system, which can be used to strongly restrict the space of possibilities. For example, a `div a b -> a / b` division function can type the variable `b` to satisfy the constraint `b != 0`. Liquid types are a special case of general refinement types, where the predicate language is a simple decidable logic, including arithmetic operations. This allows them to be encoded into SMT constraints with purely bound variables, so we can solve them with software such as Z3.

Felix makes an interesting connection between refinement types and domain-specific languages, especially in the realm of neural networks. In particular, tensor manipulation DSLs often includes things like recurrence and convolution, which introduce inductive bias compared to a vanilla linear model. These can be encoded (albeit inefficiently) in a liquid type system, where we constrain certain weights to be equal to each other, or even zero. Chris makes an interesting connection to recent work in natural language, where formalization doesn't work.

Why do we use decidable logic in refinement types? It's necessary for efficiency reasons, to guarantee termination in at least a *finite* amount of time. For example, linear integer arithmetic with addition and equality comparisons is decidable, but if you add multiplication, this suddenly becomes undecidable. Note that decidable does not necessarily mean *efficient*, as many classic NP-hard and even EXPTIME-hard (and 2-EXPTIME) problems are decidable (see LTL).[15]

22.2 Refinement Types for Synthesis [PKSL16]

Now we will briefly go over how Synquid takes refinement types into account. The brief idea is that the user input is a refinement type for the desired program, which provides some preconditions or postconditions. Synquid will then output a *provably-correct* program that matches that refinement type. For example, one possible type could be for the `replicate (n:Nat) (a:'a)` function, where we assert that the return value is a list ν such that `len` ν `= n`.

There are many, many optimizations that go into this system to make this basic idea work. The most important in practice seemed to be their *least fixpoint algorithm*, called MUSFix. I thought that this was pretty interesting. Anyway, their experiments seem to show that Synquid works (and scales) for complex programs such as synthesizing a binary heap or AVL tree.

	Benchmark	*Spec*	*SpecS*	*Time*	*TimeS*
Leon	strict sorted list delete	14	8	15.1	0.10
	strict sorted list insert	14	8	14.1	0.18
	merge sort	9	11	14.3	2.1
Jen	BST find	51	6	64.8	0.09
	bin. heap 1-element	80	5	61.6	0.02
	bin. heap find	76	6	51.9	0.38
Myth	sorted list insert	12	8	0.12	0.25
	list rm adjacent dupl.	13	5	0.07	1.33
	BST insert	20	8	0.37	0.91
λ^2	list remove duplicates	7	13	231	0.36
	list drop	6	11	316.4	0.1
	tree find	12	6	4.7	0.29
Esc	list rm adjacent dupl.	n/a	5	1	1.33
	tree create balanced	n/a	7	0.24	0.14
	list duplicate each	n/a	7	0.16	0.05
Myth2	BST insert	n/a	8	1.81	0.91
	sorted list insert	n/a	8	1.02	0.25
	tree count nodes	n/a	4	0.45	0.20

Figure 44: Comparison of Synquid "S" to well-known program synthesis tools.

The results of applying Synquid to a variety of program synthesis tasks are shown in Fig. 44,

[15] See CSE 507 for a good course on SAT solvers and decidable logic.

where *SpecS* and *TimeS* refer to the specification size and runtime of Synquid, and the other columns refer to the other tool being compared to. This demonstrates pretty good scalability. Joey mentions that another recent improvement from Nadia's group has been adding *resource types* to Synquid, which help reduce the combinatorial explosion by asserting that variables can only appear a certain number of times [KWPH19].

23 November 19th, 2020

Today, Anthony Yim will present the paper "Vid2Player: Controllable Video Sprites that Behave and Appear like Professional Tennis Players" [ZSAF20]. The stunning results of this paper are demonstrated in this video. This is our last student presentation of the semester.

23.1 Tennis Basics and Offline Annotation

Before we get started, let's give a bit of context by defining some basic tennis terms and explaining the difficulty of this graphics/vision problem. Here are some tennis terms used by the paper:

- **Far-side:** The further side of the court, away from the camera.
- **Near-side:** The closer side of the court, adjacent to the camera.
- **Offense/defense:** Which player is playing more aggressively at any given point.
- **Forehand/backend:** Generally, players are stronger on forehand and weaker on backhand.
- **Shot cycle:** The duration between when a player hits the ball, and when it comes back to them. This has two parts, each being divided into a preparation and hit phase.

The system uses offline annotation of tennis footage from professional recorded tournaments. They only consider footage from the center camera (removing instant replay and others). There are three parts to this offline annotation, part manual and part automatic:

- Human annotators add labels such as time of ball contact, time of recovery phase, shot type, shot outcome, etc., to build a comprehensive database of various tennis shot videos.
- A Mask R-CNN model is used to segment the human figures and determine their pose.
- Computer vision was also used to determine the ball's position at every frame, though this was complicated a bit by motion blur. To reduce noise in their measurements, the authors ran filtering based on a simple ODE model of ball motion.

The total amount of data is only on the order of 2–3 hours, based on a couple years of Wimbledon matches. After the offline annotation step, we have a database of many tennis video frames. The next step is synthesis, where we generate novel video frames with a degree of controllability.

23.2 Synthesis of Tennis Videos [ZSAF20]

The paper starts from a ground truth model of physics and behavioral simulation based on evaluating a recurrent function over time, then perform a specific *clip search* algorithm to find close matches with video clips in their database. In other words, they use the database to search for close neighbors with the internal state of their synthesized game, matching with offline annotations.

The player behavioral model is fairly straightforward, but it does involve a significant amount of domain-specific work. This should bear some resemblance to the tricks you typically see in 3D animation, to make convincing renders. For example, a visualization of the behavioral model is shown in Fig. 45. Notice that the authors use a kernel density estimate to model discrete points like position and ball velocity, and they also divide the tennis court into multiple regions.

One interesting point is that to actually interpolate between landing points, the authors use the forward integration of an ODE, but they need the initial velocities and positions of the ball

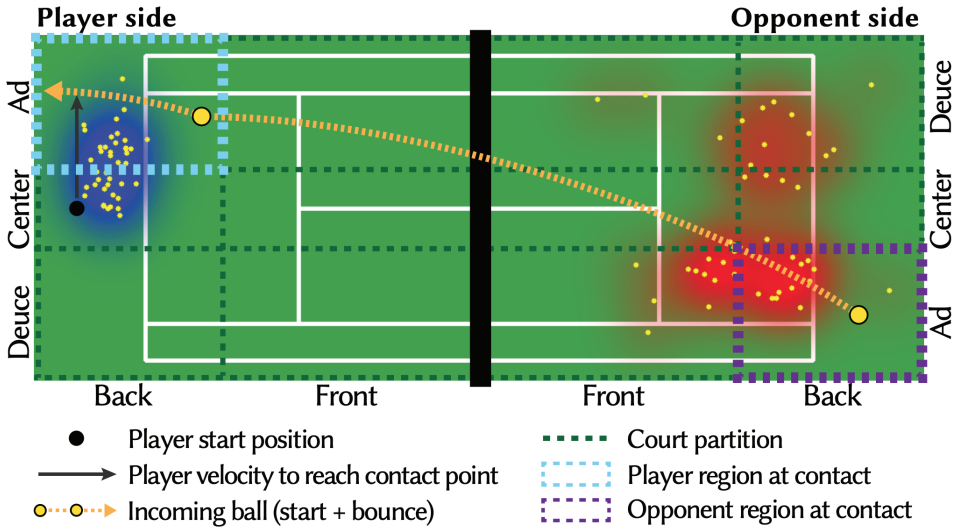


Figure 45: Visualization of shot placement distribution in behavior model.

in order to do so. How do you get these velocities? You need to solve a backward problem from the final landing position of the ball, but this involves some physics. Amusingly, the paper doesn't bother with any fancy math for this backward problem, and it just decides to grid search over all possible velocities to find the closest result, using a simple numerical integrator.

The final step of video synthesis is to take this behavioral model, adjust some parameters with filtering, and then do *clip search* through their database to look for close sprite matches. One funny issue is that sometimes, the closest match in the database may have some limbs cut off (out of the camera view frustum), so the paper plugs in some well-known image completion models like `pix2pix` to hallucinate missing limbs when necessary.

The final results are pretty surprisingly believable, even though the entire model is basically just a glorified billboarding approach where you collage 3 images (player, ball, player) onto a static background. What's interesting is that the behavioral model is very discrete, not including any details like foot movement or anatomy, which gives you a lot of high-level control. The authors demonstrate that this behavioral model actually reproduces several common "best practice" tennis behaviors seen in tournaments, such as players making less aggressive shots when in a pinch.

Some discussions indicate that a potential future step could be to do pixel prediction, where you actual synthesize *new* sprites rather than just stealing from existing video. For example, recent work at CVPR 2020 has actually produced 3D human models from a single image. This could also fix some of the issues related to choppiness when the algorithm fails to interpolate between video segments.[16] Regardless, it's quite surprising that a such a simple sprite-stitching approach yields such convincing results, and this should be a good motivation for future work.

[16] Also, this problem is only tractable because professional tennis players are extremely consistent (predictable), but it could be much harder to model something like amateur play.

24 November 24th, 2020

Today, Mark Goldstein will present a lecture on disintegration [cSR17].

24.1 Symbolic Disintegration [cSR17]

Roughly speaking, *disintegration* is a way of formalizing the ill-defined notion of conditioning on an event of measure zero. This most famously seen in the Borel-Kolmogorov paradox in probability theory. Disintegration provides a way of formalizing this notion in a PL context.

This technique is grounded in measure theory. Mark provides an extremely brief crash course in measure theory, vaguely introducing notions such as σ-algebras $\mathcal{F}$, probability spaces $(X, \mathcal{F}, P)$, and the Lebesgue integral. We'll need this background to proceed with the PL stuff later on.

The paper we're reading defines a new programming language called *Hakaru*, which uses measures (e.g., `lebesgue` and `uniform` a b) as syntactic terms.[17] It performs symbolic manipulation on these measures using operations like addition `mplus`, comparison $\leq$, and projection `fst`, `snd`. In this programming language, our key objects are *probability distributions* over real vector spaces. These are defined with denotational semantics based on integrating some mathematical function f, which is denoted by the integration operator $[[\cdot]]_I$. For example,

$$[[\textbf{uniform } 0\ 1]]_I f = \int_{[0,1]} f(x)\,\mathrm{d}x,$$
$$[[\textbf{return } a]]_I f = f(a).$$

Note that the `return` function simply represents a discrete Dirac distribution supported on $\{a\}$. This definition distills distributions into simply their behavior under integration.

The key problem that will be solved in this paper is the classical Bayesian inference task: given a joint distribution expressed as $p(a)$ and $p(a \mid b)$, compute the distribution $p(b \mid a)$ exactly. The first step will be to define integration on multivariate distributions. Given a marginal probability distribution of α, $\mu : \mathcal{M}\,\alpha$ and a conditional distribution of $\beta \mid \alpha$, $\kappa : \alpha \to \mathcal{M}\,\beta$, we have

$$[[\mu \otimes \kappa]]_I f := [[\mu]]_I(\lambda a.[[\kappa\ a]]_I(\lambda b.f(a,b))).$$

In the above equation, κ in the probability theory literature is sometimes called a *transition kernel* from α to β. This has a rigorous definition in pure measure theoretic terms. Roughly speaking, the operation being defined above intuitively looks something like

$$\iint_{\S \times \mathcal{Y}} f(x,y)p(x,y)\,\mathrm{d}x\,\mathrm{d}y = \int_{\mathcal{Y}} \left(\int_{\mathcal{X}} f(x,y)p(x \mid y)\,\mathrm{d}x \right) p(y)\,\mathrm{d}y.$$

The key idea of disintegration is that your constraint can be described as an *observation*, which provides an indexed family of random variables on a space that can be conditioned on. Then, if our observation is the last operation, we can symbolically "reverse" this outermost function application by using a *change of integration variable* (u-substitution from high school calculus), followed by a reordering of integrals using Tonelli's theorem, since our measures are nonnegative.

24.2 Algebraically Rewriting Monadic Probability Measures

The combination of marginal and conditional probability distributions described above, to form a joint distribution, can be expressed as a *monadic bind* operator. Then, the `return` function is

[17] Measures form a monad, as we will see. This idea was introduced in [RP02].

$\ll$ ("constrain outcome") : $\lceil \mathbb{M}\,\mathbb{R} \rceil \to \lfloor \mathbb{R} \rfloor \to (\mathit{heap} \to \lfloor \mathbb{M}\,\gamma \rfloor) \to \mathit{heap} \to \lfloor \mathbb{M}\,\gamma \rfloor$

$$\ll [\![u]\!]\ v\ c\ h = \bot \quad \text{where } u \text{ is atomic} \tag{48}$$
$$\ll [\![\mathbf{lebesgue}]\!]\ v\ c\ h = c\ h \tag{49}$$
$$\ll [\![\mathbf{uniform}\ r_1\ r_2]\!]\ v\ c\ h = [\![\mathbf{do}\ \{\mathbf{observe}\ \$(r_1 \le v \le r_2);\ \mathbf{factor}\ \$((r_2 - r_1)^{-1});\ \$(c\ h)\}]\!] \tag{50}$$
$$\ll [\![\mathbf{return}\ e]\!]\ v\ c\ h = \lhd [\![e]\!]\ v\ c\ h \tag{51}$$
$$\ll [\![\mathbf{do}\ \{g; m\}]\!]\ v\ c\ h = \ll [\![m]\!]\ v\ c\ [h; [\![g]\!]] \quad \text{unless } g \text{ binds a variable in } h \tag{52}$$
$$\ll [\![\mathbf{fail}]\!]\ v\ c\ h = [\![\mathbf{fail}]\!] \tag{53}$$
$$\ll [\![\mathbf{mplus}\ m_1\ m_2]\!]\ v\ c\ h = [\![\mathbf{mplus}\ \$(\ll [\![m_1]\!]\ v\ c\ h)\ \$(\ll [\![m_2]\!]\ v\ c\ h)]\!] \tag{54}$$
$$\ll [\![e]\!]\ v\ c\ h = \rhd [\![e]\!]\ (\lambda m.\ \ll m\ v\ c)\ h \quad \text{where } e \text{ is not in head normal form} \tag{55}$$

$\lhd$ ("constrain value") : $\lceil \mathbb{R} \rceil \to \lfloor \mathbb{R} \rfloor \to (\mathit{heap} \to \lfloor \mathbb{M}\,\gamma \rfloor) \to \mathit{heap} \to \lfloor \mathbb{M}\,\gamma \rfloor$

$$\lhd [\![u]\!]\ v\ c\ h = \bot \quad \text{where } u \text{ is atomic} \tag{56}$$
$$\lhd [\![r]\!]\ v\ c\ h = \bot \quad \text{where } r \text{ is a literal real number} \tag{57}$$
$$\lhd [\![\mathbf{fst}\ e_0]\!]\ v\ c\ h = \rhd [\![e_0]\!]\ (\lambda v_0.\ \lhd (\mathit{fst}\ v_0)\ v\ c)\ h \quad \text{unless } e_0 \text{ is atomic} \tag{58}$$
$$\lhd [\![\mathbf{snd}\ e_0]\!]\ v\ c\ h = \rhd [\![e_0]\!]\ (\lambda v_0.\ \lhd (\mathit{snd}\ v_0)\ v\ c)\ h \quad \text{unless } e_0 \text{ is atomic} \tag{59}$$
$$\lhd [\![-e_0]\!]\ v\ c\ h = \lhd [\![e_0]\!]\ (-v)\ c\ h \tag{60}$$
$$\lhd [\![e_0^{-1}]\!]\ v\ c\ h = [\![\mathbf{do}\ \{\mathbf{factor}\ \$((v \cdot v)^{-1});\ \$(\lhd [\![e_0]\!]\ v^{-1}\ c\ h)\}]\!] \tag{61}$$
$$\lhd [\![e_1 + e_2]\!]\ v\ c\ h = \rhd [\![e_1]\!]\ (\lambda v_1.\ \lhd [\![e_2]\!]\ (v - v_1)\ c)\ h \sqcup \rhd [\![e_2]\!]\ (\lambda v_2.\ \lhd [\![e_1]\!]\ (v - v_2)\ c)\ h \tag{62}$$
$$\lhd [\![e_1 \cdot e_2]\!]\ v\ c\ h = \rhd [\![e_1]\!]\ (\lambda v_1.\ \mathit{abs}\ v_1\ (\lambda v_1'.\ \lambda h'.\ [\![\mathbf{do}\ \{\mathbf{factor}\ \$(v_1'^{-1});\ \$(\lhd [\![e_2]\!]\ (v \cdot v_1^{-1})\ c\ h')\}]\!]))\ h \sqcup \rhd [\![e_2]\!]\ (\lambda v_2.\ \mathit{abs}\ v_2\ (\lambda v_2'.\ \lambda h'.\ [\![\mathbf{do}\ \{\mathbf{factor}\ \$(v_2'^{-1});\ \$(\lhd [\![e_1]\!]\ (v \cdot v_2^{-1})\ c\ h')\}]\!]))\ h \tag{63}$$
$$\lhd [\![x]\!]\ v\ c\ [h_1; [\![x \leftsquigarrow m]\!]; h_2] = \ll [\![m]\!]\ v\ ([\overline{[\![\mathbf{let}\ x = v]\!]; h_2}]\ ;c)\quad h_1 \tag{64}$$
$$\lhd [\![x]\!]\ v\ c\ [h_1; [\![\mathbf{let\ inl}\ x = e_0]\!]; h_2] = \rhd [\![e_0]\!]\ (\lambda v_0.\ \mathit{outl}\ v_0\ (\lambda e.\ \lhd e\ v\ ([\overline{[\![\mathbf{let}\ x = v]\!]; h_2}]\ ;c)))\ h_1 \tag{65}$$
$$\lhd [\![x]\!]\ v\ c\ [h_1; [\![\mathbf{let\ inr}\ x = e_0]\!]; h_2] = \rhd [\![e_0]\!]\ (\lambda v_0.\ \mathit{outr}\ v_0\ (\lambda e.\ \lhd e\ v\ ([\overline{[\![\mathbf{let}\ x = v]\!]; h_2}]\ ;c)))\ h_1 \tag{66}$$

Figure 46: Subset of disintegrator implementation. Key partial evaluation rules are (49) and (50).

simply the unit operation, similar to its classic behavior in Haskell. Repeated monadic binds can be interpreted denotationally as conditionally selecting random variables based on their previous values. For example, the measure

$$\overleftarrow{m}'_{2a} \triangleq \mathbf{do}\ \{x \leftsquigarrow \mathbf{uniform}\ 0\ 1;\\ y \leftsquigarrow \mathbf{uniform}\ 0\ 1;\\ \mathbf{return}\ (y - 2 \cdot x, (x, y))\}$$

describes selecting (x, y) from the uniform distribution on the unit square $[0,1]^2$, then computes $y - 2x$ conditionally as a Dirac distribution based on these concrete values. If you stare at this closely enough, this is actually the same as the measure defined implicitly below as

$$m = \mathbf{lebesgue}$$
$$M = \mathbf{do}\ \{x \leftsquigarrow \mathbf{uniform}\ 0\ 1;\\ \mathbf{observe}\ 0 \le t + 2 \cdot x \le 1;\\ \mathbf{return}\ (x, t + 2 \cdot x)\}.$$

if you are very careful to do a u-substitution to go from $t = y - 2x$ to $y = t + 2x$. The key insight of the paper is that this transformation can be done *automatically*, using a series of lazy rewrite rules in programming language terms. This is the main technical contribution, and it is implemented in a continuation-passing style. A subset of these rewrite rules is shown in Fig. 46.

We close by noting that although many integrals are very complex or perhaps intractable to compute numerically, in systems like TensorFlow, having the ability to write measures in a lazy

format is similar to having infinite lists in Haskell. They allow you to think in a high-level domain and could perhaps motivate new probability algorithms. For more on this topic, Mark recommends continuing to read papers from this Oxford-Cambridge group of researchers.

25 December 10th, 2020

Today, we have project presentations from all the students. Topics are listed below.

- Synthesizing Components of a Symbolic Physics Engine – Felix Sosa
- Adversarial robustness of models of programs: Improving the formulation of Ramakrishnan et al., 2020 to attack models of programs – Shashank Sirkant
- Heterogeneous Data Extraction using ML-Synthesis Loop – Priyan Vaithilingam
- Un-blackboxable Fairness Properties – Yafah Edelman
- Feature generating subcircuits in CNNs – Chris Hamblin
- Learning programs for graph-structured data – Alex Li
- Centaurgo – Yuan Cao
- Generative Modeling of Bach Chorales by Gradient Estimation – Eric Zhang, Romil Sirohi
- Program Induction for Feature Selection – He Sun
- Neural graph reduction, aka "running Haskell on neural nets" – Joey Velez-Ginorio
- VST proof synthesis using GamePad – Anastasiya Kravchuk-Kirilyuk
- Emergent language in multi-agent, goal-oriented environments – Anthony Yim, Adam Palay
- Modeling Moral Cognition through Program Synthesis – Andrew Kim

That concludes the seminar class! It's been an interesting virtual semester.

References

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[AP89] Barry C Arnold and S James Press. Compatible conditional distributions. *Journal of the American Statistical Association*, 84(405):152–156, 1989.

[AP20] Martín Abadi and Gordon Plotkin. A simple differentiable programming language. *PACML*, POPL, 2020.

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[BCJ+19] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20:28:1–28:6, 2019.

[BGB+16] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. *arXiv preprint 1611.01989*, 2016. Published as a conference paper at ICLR 2017.

[BRNR17] Matko Bosnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. *ICML*, 2017.

[Byr10] William E Byrd. Relational programming in miniKanren: techniques, applications, and implementations. *PhD Thesis*, 2010.

[BZSL19] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. Probabilistic verification of fairness properties via concentration. *PACML*, OOPSLA, 2019.

[CM20] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *arXiv*, 2020.

[CPSA17] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[cSR17] Chung chieh Shan and Norman Ramsey. Exact bayesian inference by symbolic disintegration. In *Principles of Programming Languages (POPL)*, 2017.

[CT19] Andrew Cropper and Sophie Tourret. Logical reduction of metarules. *Machine Learning*, pages 1–47, 2019.

[Dee19] DeepMind. Mastering Atari, Go, Chess and Shogi by planning with a learned model. *Nature*, 2019.

[DVK17] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[ED05] David J Earl and Michael W Deem. Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics*, 7(23):3910–3916, 2005.

[Ell17] Conal Elliott. Compiling to categories. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–27, 2017.

[Ell18] Conal Elliott. The simple essence of automatic differentiation. *PACML*, ICFP, 2018.

[ENP+19] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. In *Advances in Neural Information Processing Systems*, pages 9169–9178, 2019.

[EWN+20] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

[FCD15] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

[GBKT17] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. *ICML*, 2017.

[GBS+16] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv:1608.04428*, 2016.

[GKKS14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + Control: Preliminary report. *arXiv preprint arXiv:1405.3694*, 2014.

[GMR+12] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv:1206.3255*, 2012.

[GSS14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[Gul11] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[HCW+20] Ping Hu, Fabian Caba, Oliver Wang, Zhe Lin, Stan Sclaroff, and Federico Perazzi. Temporally distributed networks for fast video semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8818–8827, 2020.

[HPN17] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. DeepBach: a steerable model for Bach chorales generation. *PLMR*, 2017. Code available.

[HVU+18] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck. Music transformer. *arXiv preprint arXiv:1809.04281*, 2018.

[KBC+18] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[KGK+08] Yarden Katz, Noah D. Goodman, Kristian Kersting, Charles Kemp, and Joshua B. Tenenbaum. Modeling semantic cognition as logical dimensionality reduction. *CogSci*, 2008.

[KMP+18] Ashwin Kalyan, Abhishek Mohta, Alex Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *ICLR*, 2018. See also blog post.

[KWPH19] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–268, 2019.

[Lom06] Tania Lombrozo. The structure and function of explanations. *Trends in cognitive sciences*, 10(10):464–470, 2006.

[MAT13] Jonathan Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.

[MDK+18] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. DeepProbLog: Neural probabilistic logic programming. *NeurIPS*, 2018.

[MTG+13] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195, 2013.

[NWA+19] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. *arXiv preprint arXiv:1909.12252*, 2019.

[OSTG11] Timothy O'Donnell, Jesse Snedeker, Joshua Tenenbaum, and Noah Goodman. Productivity and reuse in language. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 33, 2011.

[OZ15] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015. Code is available.

[PEK+20] Yewen Pu, Kevin Ellis, Marta Kryven, Josh Tenenbaum, and Armando Solar-Lezama. Program synthesis with pragmatic communication. *arXiv preprint arXiv:2007.05060*, 2020.

[PKSL16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[PMS+16] Emilio Parisotto, Abdelrahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv:1611.01855*, 2016.

[PS20] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

[RHW+20] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043*, 2020.

[RN02] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson Education, 2002.

[RP02] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 154–165, 2002.

[SAB+19] Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. Formal verification of higher-order probabilistic programs. In *Principles of Programming Languages (POPL)*, 2019.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering Chess and Shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[SLH+20] Alexey Svyatkovskoy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. *arXiv preprint arXiv:2004.13651*, 2020.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[SSTS20] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. *arXiv preprint arXiv:2007.02220*, 2020.

[Sun06] Cass R Sunstein. Two conceptions of procedural fairness. *Social Research: An International Quarterly*, 73(2):619–646, 2006.

[TBM+19] Zenna Tavares, Javier Burroni, Edgar Minasyan, Armando Solar Lezama, and Rajesh Ranganath. Predicate exchange: Inference with declarative knowledge. In *International Conference on Machine Learning (ICML)*, 2019. Code is available.

[TdSB+19] Philip S. Thomas, Bruno Castro da Silva, Andrew G. Barto, Stephen Giguere, Yuriy Brun, and Emma Brunskill. Preventing undesirable behavior of intelligent machines. *Science*, 366(6468):999–1004, 2019.

[TKGG11] Joshua B Tenenbaum, Charles Kemp, Thomas L Griffiths, and Noah D Goodman. How to grow a mind: Statistics, structure, and abstraction. *science*, 331(6022):1279–1285, 2011.

[TSC20] Andrew Tao, Karan Sapra, and Bryan Catanzaro. Hierarchical multi-scale attention for semantic segmentation. *arXiv preprint arXiv:2005.10821*, 2020.

[VBC+19] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[vdMPYW18] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. *An Introduction to Probabilistic Programming*. arXiv, 2018.

[vKAvH20] Emile van Krieken, Erman Acar, and Frank van Harmelen. Analyzing differentiable fuzzy logic operators. *arXiv preprint arXiv:2002.06100*, 2020.

[VT20] Elena Voita and Ivan Titov. Information-theoretic probing with minimum description length. *arXiv preprint arXiv:2003.12298*, 2020.

[ZDW+19] Daniel Zheng, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *PACML*, ICFP, 2019.

[ZSAF20] Haotian Zhang, Cristobal Sciutto, Maneesh Agrawala, and Kayvon Fatahalian. Vid2player: Controllable video sprites that behave and appear like professional tennis players. *arXiv preprint arXiv:2008.04524*, 2020.

[ZZSE16] Shengjia Zhao, Enze Zhou, Ashish Sabharwal, and Stefano Ermon. Adaptive concentration inequalities for sequential decision problems. In *Advances in Neural Information Processing Systems*, pages 1343–1351, 2016.