Input:
Parameters:
Output:
SE[DOWHILE]DodoWhile[1] 1

Boolean Function Synthesis using Gated Continuous Logic Network

Aditya Kanade, Chiranjib Bhattacharyya, Deepak D'Souza, Ravi Raja, and Stanly Samuel

***Abstract*— Boolean Function Synthesis is a fundamental problem in computer science with lots of different applications. The state of the art tool is able to solvel 356 out of 609 benchmarks, leaving room for improvement. We are using a specialized Neural Network called Gated Continuous Logic Network to synthesize the formulae that satisfies the given specification. Our expectation from using a Neural Network is two fold: 1. To beat the state-of-the-art tools and 2. To study whether a Neural Network can capture the underlying semantics of the Boolean Formula Specification.**

I. Introduction

The problem of Boolean Function Synthesis is a well known problem in computer science and has been in the limelight for past decade. Recently, synthesis of Boolean functions has found its applications in wide range of areas which includes reactive strategy synthesis [1], certified QBF-SAT solving [2], automated program synthesis ([5], [6]), circuit repair and debugging [3] and the likes. This has motivated the community to develop practically efficient algorithms for synthesizing Boolean functions. Latest tool Manthan [4] claims to have beaten all the other state of the art tools by a margin of 76 benchmarks. Manthan uses a Decision Tree based Learning approach to generate the skolem functions satisfying the given specification.

Based on this exhaustive survey, we propose the following problem statement:

Problem Statement: Given a logical specification and a DSL for the space of programs, we wish to design a system that synthesizes sketches in such a way that the work load between neural synthesis and symbolic search is managed efficiently i.e. we want a neural network to synthesize sketches when pattern based techniques are unable to proceed and invoke a solver to complete the sketch. We want the neural network to be trained using a multi-modal "concolic" approach i.e by using an input embedding based on logical specifications and I/O examples generated from these I/O examples.

To the best of our knowledge, we are the first to work on sketch generation using logical embeddings.

Brainstormed under the helpful guidance of Aditya Kanade, C. Bhattacharyya, and D. D'Souza

Aditya Kanade, C. Bhattacharyya, D. D'Souza, R. Raja and S. Samuel are with IISc, Bengaluru, India.

Other works using sketches: Neural Edit Completion is a code completion technique that uses the context of the code environment as specification. A sketch in this context is the partial code that the user provides.

Program synthesis using conflict-driven learning also talks about sketches in terms of partial programs.

Contribution: In this report, we will talk about a research proposal and preliminary work in this direction. The major topics are:

- A line of direction to generate synthetic dataset for SyGuS benchmarks.
- A "concolic" neural network architecture using logical embeddings and I/O embeddings.
- A prototype tool SyGuS-Sketcher using existing techniques with experimentation plan.

II. Motivation

One of the motivating examples that we had looked into involved the Conditional Linear Integer Arithmetic track of SyGuS benchmarks. A state of the art solver CVC4 was not able to solve the following constraint: $x\ is\ even \implies div(x) + div(x) = x$ where $div : Real \rightarrow Real$ is the function to be synthesized. The constraint hints on the fact that the division operator needs to be used in the final program. A human would have had no issue in figuring this out. Consequently, we expect to design and train a neural network that could pick up these patterns and synthesize sketches accordingly. For example, feeding this constraint to the neural network could synthesize a sketch as follows: $div(x) = x/HOLE$ i.e. the neural network realizes that the division operator is to be used but doesn't really know which constant is needed so it synthesizes a hole in place of the constant. A traditional sketch solver can then be used to synthesize the constant 2.

On the other hand, the neural network may even choose to synthesize the sketch $div(x) = HOLE/HOLE$ in which case you will also need an enumerative solver to fill up the hole for the variable x.

The other motivation is to improve the performance of existing tools by the use of sketches. In the experiments section, we will comment on DeepSynth's performance on its benchmarks.

III. Background

The required background for this report are CEGIS [?], CEGIS(T)[?], CEGNS[?] and the SyGuS [?] benchmarks. For brevity, we refer the readers to the above papers.

However, we explain DeepSynth's Neural Network architecture as background required for our proposal.

A. DeepSynth's Neural Network Architecture

This part explains the neural network architecture and training data generation techniques used in deepsynth and how are these integrated together.

1) Neural Network: The DeepSynth Neural Network, shown in Fig. 1 is made up of several parallelised sequence-to-sequence neural networks, each of which processes a single counterexample (i.e. I/O example) and computes a probability distribution over the likely sequence of tokens in the program given that input/output example. The outputs from all the sequence-to-sequence networks are combined using pooling. In the following sections we describe in detail a single sequence-to-sequence network, and then the pooling process.

a) Processing a Single counterexample: The Seq2Seq network used for processing a single counterexample is shown in Fig. 2 and consists of:

1) An **input encoder** LSTM network, which receives the input argument values from the I/O example.
2) An **output encoder** LSTM cell, which receives the output value from the I/O example and is conditioned on the final cell and hidden states of the input encoder network.
3) A **synthesis decoder** LSTM network, which is trained on the **target program** and conditioned on the final output encoder state.

Input arguments are fed into the input encoder network as a sequence. The cell state and hidden states of all three LSTM cells (for the input encoder, output encoder, and synthesis decoder respectively) are set to be 128-dimensional vectors. Program tokens are represented using a learnable 128-dimensional vector embedding. Every program token in the token vocabulary has a 128-dimensional vector representation which is updated at each time step of training.

DeepSynth evaluates three different representations for input arguments and the output values, which is referred as **input-modes:**

1) **"normalised"** mode, where every parameter value p is represented by a normalised scalar value, more specifically $\frac{p}{2^{31}} - 1$
2) **"binary"** mode, where p is provided in its binary representation, a 32-dimensional vector of 0s and 1s. This representation is more useful for helping the network detect minute differences between inputs.
3) **"normBinary"** mode, which combines both normalised and binary representations, thus producing a 33-dimensional vector representation for every input argument and the output value.

Finally, it includes an additional setting for the network, in which the synthesis decoder is provided with the arity (i.e. number of input arguments) of the target program in addition to the target program's tokens.

b) Pooling: To process n I/O examples, the DeepSynth neural network creates n identical copies of the single-IO Seq2Seq network, with each receiving one of the n I/O examples. Once it has the encoded I/O examples, it is then fed to n synthesis decoders along with a GO token at time step 1, and a 128-dimensional hidden state is produced from each decoder. Let o_{t_i} be the hidden state produced at time t by synthesis decoder i.

To aggregate all hidden states o_{T_i} at any given time T , these values are first passed through a **fully-connected neural network layer** to compute n values fc_{T_i} such that $fc_{T_i} = tanh(W * o_{T_i} + b)$, where W is a 128-by-128 weight matrix and b is a 128-by-1 bias matrix. Following this, the n fc_{T_i} values are aggregated into a unique 128-dimensional aggregate output value O_T using the max pooling operation as follows:

$$O_T[j] = \max_{1 \leq i \leq n} fc_{T_i}[j]$$

where $O_T[j]$ is the j^{th} dimension of O_T and $fc_{T_i}[j]$ is the j^{th} dimension of fc_{T_i}. Finally, O_T is used to compute the probability distribution D_T over the next token using a softmax neural network layer. More formally, $D_T = softmax(W_{Lin} * O_T + b_{Lin})$, where W_{Lin} is a 50-by-128 weight matrix and b_{Lin} is a 50-by-1 bias vector (50 is the size of vocabulary). The pooling procedure is shown in Fig. 3

2) Training Data Generation: The Training Data consists of a set of Candidate programs, each accompanied by a corresponding set of input/output examples. For program synthesis, neural networks typically require millions of training examples therefore, in order to obtain a sufficiently large training data set, they designed a program generator that randomly generated programs made up of syntactically correct combinations of SYGUS-IF instructions [1]. Randomly generated data set also consists of **"bad training data"** which is then pruned out using some set of rules and SMT based procedure. **Bad Training Data:** Redundant programs and in-

[1] https://sygus.org/assets/pdf/SyGuS-IF_2.0.pdf

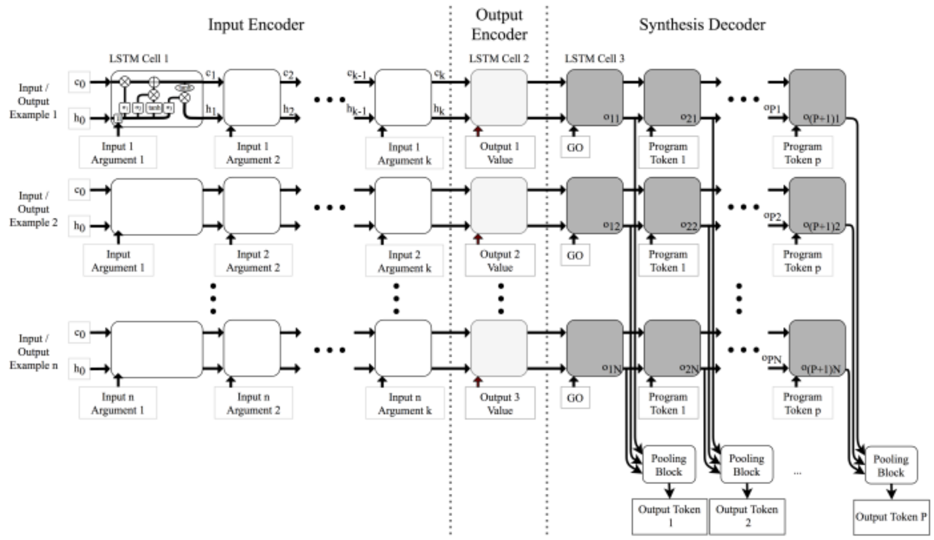

Fig. 1. The DeepSynth Neural Network

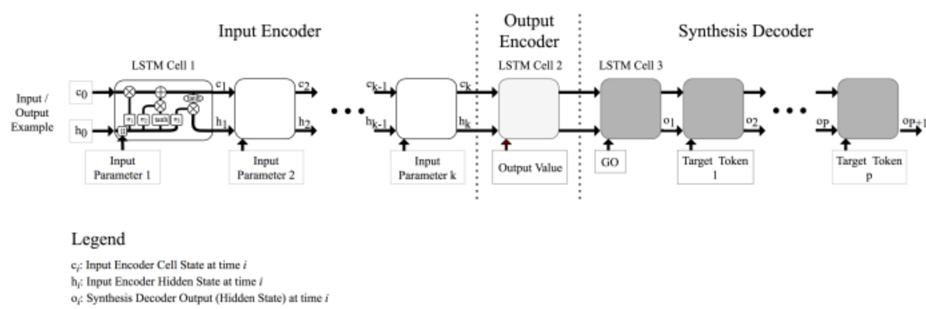

Fig. 2. Seq2Seq network for a single I/O example

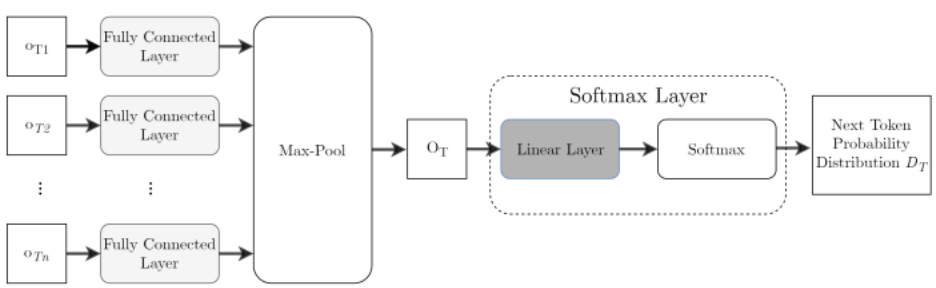

Fig. 3. The Pooling Operation

put/output examples that do not sufficiently differentiate between programs. These are removed using following methods.

a) Eliminating redundancy in programs: **Constants in programs** There are several cases where the choice of constants in programs can introduce redundancy.

- Shifting a bit vector by any number greater than the bit vector width produces a 0. Thus, a rule is introduced such that no generated program contains any shift operation with the second operand greater than the width of the first.
- Zero value constants are disallowed in all cases since they almost always introduce redundancy.

SMT based Redundancy Check SMT-solver is used to identify two sources of redundancy:

- An if-then-else statement is redundant when it always returns one of its conditional outputs. This redundancy is identified by recursively identifying leaf operators (i.e. operators not having nested ite statements) and checking their branch satisfiability with an SMT solver. If a branch is unreachable, the if-then-else statement is replaced with the other branch and call the recursive function again. This is continued until no leaf statements are redundant.
- The SMT Solver also identifies those programs that always return a single constant value, or always return a single one of the input arguments, and remove them from the training set.

b) Input/Output Example Generation: Input/Output examples are generated uniformly from a target program by executing the programs in order to get the corresponding outputs.

DeepSynth generates Input/Output (I/O) examples for each random program such that this I/O is informative with respect to this program, i.e., the I/O must cover all conditional branches in a program so as to fully portray a program's semantics. This is done by using a combination of randomly generated inputs and SMT solving. For a program P with C conditional statements and a shift assertion set S, "Smart" mode cycles through all 2^C possible execution cases and, where a case is satisfiable subject to S, it produces a new I/O example. This is done until N I/O examples are produced. In order to obtain a distribution of inputs over the input space when using an SMT solver, Z3 is used with the phase selection set to random.

c) Program Tokenisation: Programs are converted into a **token** representation that the network can process. This representation encodes every operator in the DSL, as well as up to 10 different input parameters in a program, as its own token. Constants are each encoded using 8 tokens, such that each token represents a 4 bit value and can take on one of 16 values. Two extra tokens are introduced for GO and EOS, which is used by the network during training to learn when synthesis starts/stops. To uniformise the length of its program sets, a PAD token is introduced. In total the vocabulary consists of 50 tokens for representing operations, input parameters, and constants that can appear in the program DSL.

d) Batches: Programs' token representations and I/O examples are aggregated into batches of size B. It also pads batch programs to one same length for computational reasons and converts I/O values into normalised and binary input format.

IV. Research Proposal

In this section, we discuss our research idea in detail. We first discuss the overall idea of our research using a block diagram. We then discuss a neural network architecture that we propose for the problem. We also explain DeepSynth's NN architecture for clarity.. Finally, we conclude with a discussion on how to generate a dataset to train this problem. The benchmarks that we are looking into are the standard SyGuS benchmarks.

A. Overall idea

Figure 4 describes the high level idea for synthesizing sketches from logical embeddings. We first sample a finite set of I/O examples from the logical specification. We wish to have a pre trained neural network for a given DSL that is trained to generate the required sketches when a logical specification and I/O specification are given as input. As a novel extension, it may also be possible for the NN to take as input a bad sketch and make decisions accordingly. This would be something similar to CEGIS(T) but using an NN. We then feed it to a traditional solver that solves the sketch using enumerative techniques if the sketch has non constant holes and constraint based techniques, if the sketch has constant holes. If the sketch is infeasible, then we use this to direct our NN to synthesize a better sketch. The meaning of an infeasible sketch is that there does not exist a valid (w.r.t. the DSL) completion of the sketch for the given specification.

The major research question here is how to make the NN understand whether a given sketch is an infeasible sketch or not.

B. Proposed Neural Network Architecture

This section details about the proposed Neural Network architecture for our work as shown in Figure 5. We use Neural Network for two purposes - 1) Learning representation, and 2) Synthesizing Sketches from the learned representations. These are elucidated below:

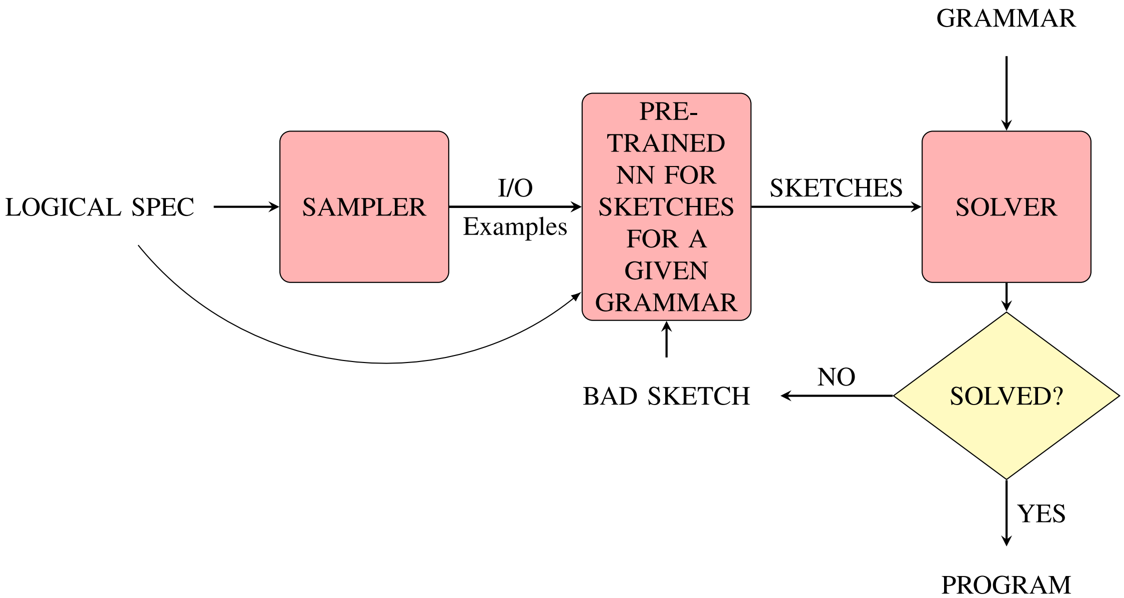

Fig. 4. Synthesizing Sketches from Logical Embeddings

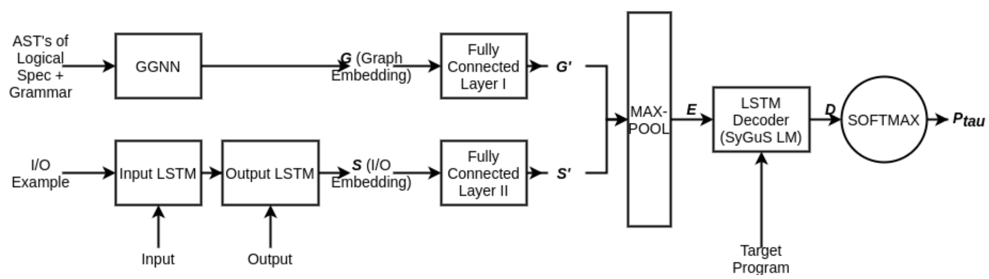

Fig. 5. Proposed Neural Network Architecture. The dimensions of all the hidden state vectors $\mathcal{G}$, $\mathcal{S}$, $\mathcal{G}'$, $\mathcal{S}'$, $\mathcal{E}$, $\mathcal{D}$, is 128-by-1 and size of P_τ is equal to the size of Vocabulary. The Encoder gives output as $\mathcal{E}$, which is then passed to the Decoder LSTM to get the program token probabilities P_τ

1) Neural Network Architectures: This section explains different architectures to be used in this work.

a) RNN: Recurrent Neural Networks or RNN (Fig. 6) are generally applied to sequential data. Given a Sequence of input (x_1,....,x_T), an RNN computes the sequence of outputs (y_1,....,y_T) by iterating the following equation:

$$h_t = sigmoid(Ux_t, Vh_{t-1})$$
$$y_t = Wh_t$$

where h_t represents hidden state of RNN.

The RNN can easily encode sequences to sequences whenever the alignment between the inputs the outputs is known ahead of time (i.e both having same lengths). However, the application of an RNN to problems whose input and the output sequences have different lengths is not very clear. Cho et al. [**?**] uses one RNN to encode input to a fixed-size vector and uses another RNN to generate output sequence from the fixed-size vector. Theoretically this approach should work but it is seen practically that it fails to recall long-term dependencies ([**?**], [**?**]). However, LSTM's [**?**] are able to capture the long term temporal dependency in the sequence.

b) Gated Recurrent Unit: Gated Recurrent Unit (Fig 7) or GRU cell is an RNN cell with gates for controlling the amount of information to be retained from the past. Initially, for t=0, the output vector is $h_0 = 0$. [2]

[2] https://en.wikipedia.org/wiki/Gated_recurrent_unit

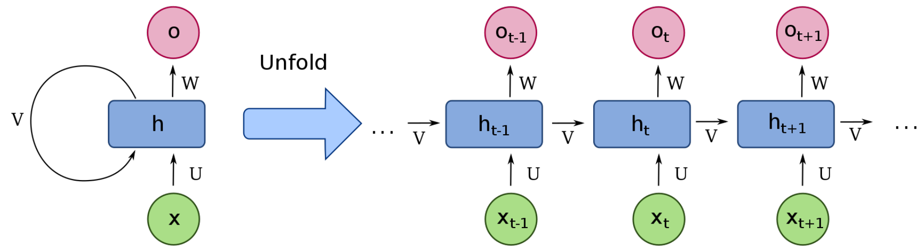

Fig. 6. Recurrent Neural Network (RNN) Cell x_t: input vector, h_t: hidden state vector, o_t: output vector, W, U and V: parameter matrices

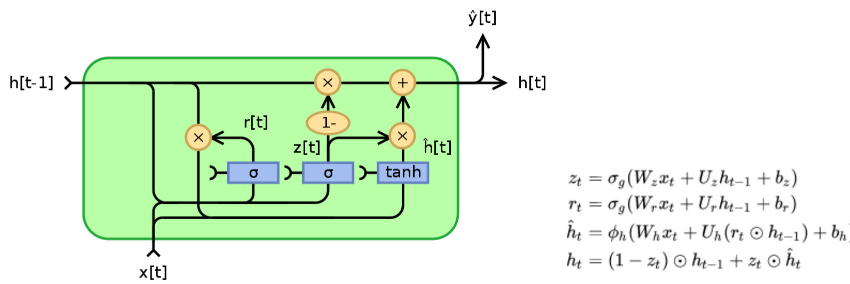

Fig. 7. Gated Recurrent Unit (GRU) Cell, Propagation Model (x_t: input vector, h_t: hidden state, $\hat{h_t}$: candidate activation vector, z_t: update gate vector, r_t: reset gate vector, W, U and b: parameter matrices and vector, σ_g: sigmoid, ϕ_h: tanh

c) LSTM: LSTM estimates the conditional probability $p(y_1, ..., y_{T'}|x_1, ..., x_T)$ where $(x_1, ..., x_T)$ is an input sequence and $y_1, ..., y_{T'}$ is an output sequence whose length $T^{'}$ may be different from the length of input sequence. The LSTM computes this conditional probability by first by learning a fixed-dimensional representation v of the input sequence $(x_1, ..., x_T)$ which is the last hidden state of the LSTM. This representation vector v is then passed to another LSTM, as an initial hidden state, that serves as a Language Model (trained over the Grammar of required output language) to compute the probability of $y_1, ..., y_{T'}$

$$p(y_1, .., y_{T'}|x_1, .., x_T) =$$
$$\Pi_{t=1}^{T^{'}} p(y_t|v, y_1, .., y_{t-1})$$

Where $p(y_t|v, y_1, ..., y_{t-1})$ distribution is represented with a softmax operation over the vocabulary (unique tokens in the output grammar G). The first LSTM is the Encoder part of a Seq2Seq architecture while the second one is the Decoder part. Fig. 8 shows a schematic diagram of an LSTM Cell.

d) Gated Graph Neural Network (GGNN): We use Gated Graph Neural Network (Fig. 9) (Li et al., 2015)[**?**] for representing specifications. This part gives a brief understanding of how GGNN works. Let G = (V, E, X) be a graph with V being set of vertices, E = (E_1, E_2, ..., E_k) be the set of directed edges where k is the number of edge types, and X is the feature set of nodes. In our case k=2 types of edges viz. intrinsic and extrinsic depending upon whether it is connecting nodes internal to grammar or logical spec or is it connecting the grammar and logical spec graphs together. Each vertex v $\in$ V is annotated with feature vector $x^{(v)} \in R^d$ representing the features of node (e.g. embedding of node label).

Each of the node is associated with a state embedding vector $h^{t^{(v)}}$ at time step t, initialized with node feature vector $x^{(v)}$. The sizes of State vector and feature vector are kept same (R^d). Information is propagated among each nodes of the graph by Message Passing. Each node v receives messages of type k from its neighbours u, where each message is computed from its current state vector as $m^{t^{(u)}}_k$ = $f_k(h^{t^{(u)}})$. Here, f_k can be an arbitrary function which we choose to be a Linear Layer that transforms the state vector with a dxd weight matrix W as $f_k(h^{t^{(u)}}) = (W \cdot h^{t^{(u)}})$. All the state vectors are updated at same time by aggregating all the messages from its neighbours as $M^{t^{(v)}}$ = g($m^{t^{(u)}}_k$) at time step t, where g is an aggregation function which is element-wise summation in our case. A new state vector $h^{t+1^{(v)}}$ for each vertex v at next time step t+1 is calculated as

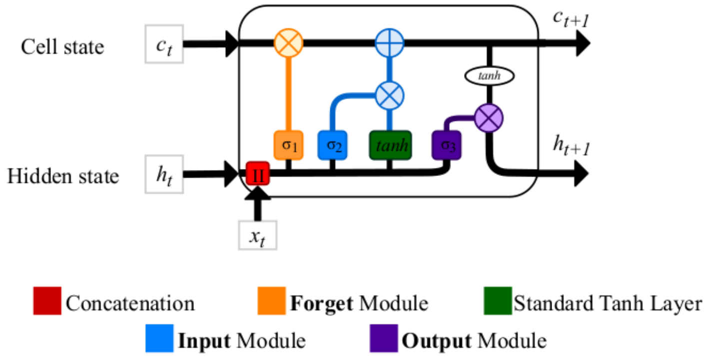

Fig. 8. Long-Short Term Memory (LSTM) Cell

Fig. 9. (a) An Example Gated Graph Neural Network (GGNN), (b) Message Passing

$h^{t+1(v)}$ = GRU($M^{t(v)}$, $h^{t(v)}$), GRU is Gated Recurrent Unit (Cho et al.) [**?**]. These updations are carried out for a fixed time step T and the final state vectors are used as node representation. To get a graph level representation $\mathcal{G}$ we do max pooling of final node representations as $\mathcal{G}$ = MAXPOOL($h^{t(v_i)}$) for each node v_i.

e) Seq2Seq Architecture: It is an Encoder-Decoder model, where the encoder encodes the input sequence (I/O Examples in our case) into an embedding vector which is passed as input to the Decoder network. The Decoder then decodes the output sequence (Sketch in our case). We use Long Short Term Memory (LSTM) both for Encoding and Decoding.

2) Learning Representations: The Constraints or Semantic/ Logical Specification and the Grammar or Syntactic Specification are highly structured, which is missed when we consider them just a sequence of tokens. The structure contains important information like order of operators in the constraints or derivations of the grammar. However, it may fail to see the semantic similarity of syntactically different constraints. Consider the following three constraints -

(1) ($\implies$ (= (mod x 2) 0) (= (+ (div1 x) (div1 x)) x))

(2) (not ($\implies$ (= (mod x 2) 0) (= (+ (div1 x) (div1 x)) x)))

(3) ($\implies$ (True) ($\implies$ (= (mod x 2) 0) (= (+ (div1 x) (div1 x)) x)))

Assume same grammar G for all three constraints. Constraints are said to deliver the semantics of the function to be synthesized but still due to its symbolic nature, constraints (1) and (2) are considered syntactically much more similar than constraints (1) and (3) despite the fact that (1) and (3) are essentially asking for same function i.e division by 2.

While I/O examples can encode concrete functional behaviour of intended function to be synthesized. For e.g. (4, 2), (10, 5) I/O pairs clearly concludes that (1) and (3) are semantically equivalent. But I/O examples suffers from insufficient path coverage. And even for covered paths, it requires huge dataset to achieve generalization, leading to expensive training procedure.

Therefore we aim to benefit from both kinds of specifications viz. SMT and I/O examples. For this we present a novel approach for learning representations of both SMT spec and I/O spec. This idea is based on Learning Blended, Precise Semantic Program Embeddings by Wang et al. [**?**]

a) Encoder: Learning Representations for Logical spec and given Grammar: In order to retain the structural properties of constraints and grammar we construct GGNN for the underlying AST's. Each node of the GGNN is a GRU cell and each edge is a feed forward network. It is trained and the final graph embedding $\mathcal{G}$ are obtained as explained in IV-B.1.d. Our architecture for learning representations for syntactic and semantic specification is similar to the one used by Xujie Si [**?**] which joins the graphs of logical specification and the grammar together as one graph. In addition, we extend it for General Track of SyGuS Competition.

b) Encoder: Learning Representations for I/O spec: We use an **input LSTM encoder** and an **output LSTM encoder** as described in deepsynth [**?**]. Each argument of the input(I) of an I/O example is fed to the input LSTM. The final hidden state of the input LSTM encoder becomes the initial hidden state of the output LSTM encoder and the output (O) is fed as input to it. The final hidden state of the output LSTM is the learned representation of one I/O example. Embeddings for all the I/O examples of a program are aggregated together to get the final I/O embedding $\mathcal{S}$.

c) Aggregating $\mathcal{G}$ and $\mathcal{S}$: Finally we learn a combined representation for both logical spec and the I/O spec by aggregating the two embeddings $\mathcal{G}$ and $\mathcal{S}$ by first passing them through a fully connected neural network and then performing max pooling operation on them. We denote this final embedding by $\mathcal{E}$.

3) Decoder: Neural Network for Synthesizing sketches: Final embedding $\mathcal{E}$ from the Encoder is then passed to an LSTM decoder, which is a Language Model trained over SyGuS Grammar, to produce final hidden representation $\mathcal{D}$. We pass this embedding $\mathcal{D}$ through a Softmax layer to get the token probability distribution P_τ where τ denotes the set of tokens. The Decoder at each time step t predicts the next most probable token τ_i. τ_i is given as input to the LSTM decoder for the next time step. At each time step token with highest probability is generated. This is called Greedy Decoding technique. Greedy Decoding is an irreversible process i.e. if a wrong token is generated then it will lead to a wrong sketch, there's no way to backtrack. Therefore, we use Beam Search Decoding, which searches for $\mathcal{K}$ most probable tokens at each time step and generates $\mathcal{K}$ most likely sketches.

4) Solving Sketches and Training the Neural Network: The sketches generated by the Decoder is given to a Solver which finally generates $\mathcal{K}$ complete programs $P_\mathcal{K}$. These are then compared with the true program $\mathcal{P}$ and cross-entropy loss is calculated as follows:

$$\mathcal{L} = -\sum_{i=1}^{d} \mathcal{P}_i . log(P_{\mathcal{K}_i})$$

where $P_\mathcal{K}$ and $\mathcal{P}$ are 128 dimensional vectors denoting the output programs and the ground truth programs. $\mathcal{P}_i$ and $P_{\mathcal{K}_i}$ are the i^{th} dimension value of $\mathcal{P}$ and $P_\mathcal{K}$ respectively. We train our network by the method of *teacher forcing*, in which the target sequence is fed to the Decoder directly during the training and used to compute the next token. The final program P_{final} obtained by minimizing the loss $\mathcal{L}$, is our solution program.

C. Generating training data

Our aim is not to replace IO with generalized specifications but to aid these specifications with I/O examples. Generating training data has been the bottleneck for this research work since it is not a trivial pursuit in our case. Consequently, if we manage to generate synthetic SyGuS training data, it can prove to be a substantial contribution for Deep Learning based SyGuS research. Thus, generating data is one of the ideas we will focus on in this report. As discussed in Section III-A, DeepSynth's Neural Network architecture requires training data which is a pair of finite I/O specification and correct program. They achieve this by first randomly generating programs and inputs with certain restrictions. Next, they generate the outputs by feeding these inputs to the generated programs. Thus, it is relatively easier to generate programs this way. Moreover, they claim to have promising results using this strategy. This is encouraging for us to try a similar strategy for the generation of synthetic sygus benchmarks as well. However, we need training examples of the form (logical constraint + IO examples, correct program). The only option is to generate programs by feeding the constraints to different solvers. A natural question to ask would be how to generate millions of constraints? This is a challenging problem and will require a new data generation pipeline. We will discuss some ideas for data generation in this section. Our aim is to train the Neural Network with constraint-program pairs in such a way that the NN can generalize well. Thus, it is important that we feed various combinations of the existing constraints to the NN.

Mutations of existing SyGuS benchmarks: We can exploit properties of logical formulae such as commutativity, associativity etc to synthesize semantically equivalent but syntactically different constraints. For example, the formula A and B can be represented as B and A. Such mutations to existing formula will help in generation of constraints. Moreover, it should generate the same program which will aid the training of the NN.

Using different solvers: The other possibility is to use different solvers so that they generate different programs for the same constraint. We have made this observation while experimenting with different tools. Correctness of these programs is important as well and hence a reliable solver such as CVC4 is important. As per our experiments, some solvers such as DryadSynth were not sound.

Bounding synthesis time: Assume we require 80 million training examples as per DeepSynth. In such a case, we need 80 million constraints randomly generated using an intelligent strategy. Assuming we have access to a standard solver that can solve each constraint in approx 10 ms (which is a realistic expectation), the solving should take approx 10 days on a basic machine whereas increasing it to 50 ms may need 52 days of solving and 1 sec bound leads to 2.5 years of solving. Thus, bounding the synthesis time for constraints is important. More importantly, it is important to generate constraints that will be solvable within these bounds.

Random generation of constraints: The previous ideas may not scale to millions of training data that we need. Thus, at some point, we have to look into generating random constraints using ideas from DeepSynth as a starting point. This becomes challenging in our situation as the structure of the constraints change for different classes of SyGuS benchmarks. For example, Loop Invariant SyGuS benchmarks have a PRE, TRANS, POST format whereas the others do not. Thus, it is prudent to decide a class of benchamarks first and proceed.

Number of training examples: Since we are using logical constraint and I/O specification as an input to the NN instead of just I/O specifications, it may be possible that we require fewer examples for training. However, this can only be empirically evaluated.

V. SyGuS-Sketcher: A preliminary prototype

Instead of solving the larger problem as mentioned in the previous section, we decided to solve a sub-problem first. We have begun implementing a preliminary prototype as a tool which we call SyGuS-Sketcher. SyGuS-Sketcher is built atop DeepSynth [?] and is available on github.[3]. In this section, we discuss the tool architecture for SyGuS-Sketcher and experimentation direction. The aim of this prototype is to show that sketches for logical specifications can indeed help improve the performance and number of benchmarks solved as compared to DeepSynth. This is our hypothesis.

A. Tool architecture

Figure 10 shows the prototype architecture. As you can see, the architecture is similar to DeepSynth except for two modifications. First, out of the top K candidates selected by the DeepSynth's NN, we select the candidate most likely to be correct, which is a complete program. Next, we replace the constant values with constant literals to generate a sketch. This is as good as assuming that we do not trust the constants generated by the Neural Network but use existing sketch compilers to fill up the program while guaranteeing the correctness of the specification. Thus, this boils down to a problem of solving sketches with constant holes which is exactly what the Sketch tool[?] does. However, for the bit-vector invariant generation benchmarks that we use, Sketch has limited support. For example, bit-vector subtraction and relational operators are not supported in Sketch due to which we had to look for other options. Thus, we looked into CEGIS(T) verifier which has this support. They have a Fourier Motzkin and an SMT based verifier and we need to decide which one fits in our case. We are currently in the process of implementing this integration. This is the second modification from DeepSynth.

[3]`https://github.com/stanlysamuel/sygus-sketcher`

B. Planned experiment

We plan to test this prototype against DeepSynth's 88 bit-vector invariant benchmarks. We describe a subset of these benchmark timings below:

Benchmarks	DeepSynth	SyGuS-Sketcher
anfp-new	23.7529s	.
formula22	44.5536s	.
hola.05	165.086s	.
array-new	76.1318s	.
cegar2-new	106.402s	.

As you can observe, DeepSynth takes a considerable amount of time whereas traditional solvers without neural network architectures have an order in milliseconds. This is where we expect Sketches to play a role in improving the performance time.

VI. Future work and extensions

We wish to achieve the following in the coming weeks:

- Complete the implementation of SyGuS-Sketcher and compare it emprically with DeepSynth.
- Generate dataset as explained and train the proposed NN model for invariant benchmarks and perform relevant experimentation.
- Extend to more classes of SyGuS benchmarks such as CLIA to check if this architecture can fill the gaps, as mentioned in the motivating example.

Research directions:

If a sketch generated by the neural network is infeasible (i.e. the solver is not able to find a valid completion of the sketch that satisfies the specification), how do we communicate it back to the neural network? More specifically, how do we leverage a neural network to understand infeasible sketches, given an embedding of a bad sketch using a GGNN? A deduction guided RL approach [?] uses a similar idea but to update the policy of an RL network when an infeasible partial program is returned.

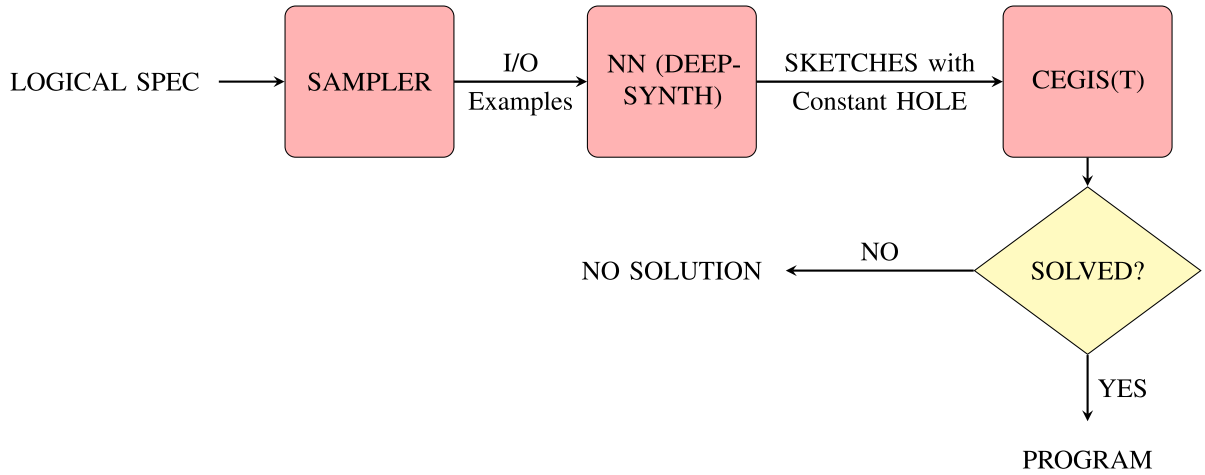

Fig. 10. SyGuS-Sketcher architecture

The verifier that SyGuS-Sketcher uses can only synthesize constant holes. Consequently, infeasible sketches can be encoded as logical formulae with constant literals and fed as a constraint back to the synthesizer. However, if the hole is not a constant hole, how do we regard it as a bad sketch is still not clear. One plausible direction is to look into the use of Conflict Driven Learning techniques for the same.

VII. Conclusion

In this report, we have proposed a research idea and a plausible line of attack. We have also started preliminary work in implementing the prototype and experimental evaluation is under way.

This report highlights the following contributions: 1) A line of direction to generate synthetic dataset for SyGuS benchmarks, 2) A ”concolic” neural network architecture using logical embeddings and I/O embeddings and 3) A prototype tool SyGuS-Sketcher using existing techniques with experimentation plan.

References

[1] Alur, R., Madhusudan, P. Nam, W. Symbolic computational techniques for solving games. 118–128 (2005).

[2] Balabanov, V., Jiang, JH.R. Unified QBF certification and its applications. 2012.

[3] Jo, Satoshi and Matsumoto, Takeshi and Fujita, Masahiro. SAT-Based Automatic Rectification and Debugging of Combinational Circuits with LUT Insertions. In *2012 IEEE 21st Asian Test Symposium*, pages 19–24, 2012.

[4] Priyanka Golia and Subhajit Roy and Kuldeep S. Meel. Manthan: A Data Driven Approach for Boolean Function Synthesis. *CoRR*, abs/2005.06922, 2020.

[5] Solar-Lezama, A. Program sketching. *Int J Softw Tools Technol Transfer*, 15(475–495), 2013.

[6] Srivastava, S., Gulwani, S. Foster, J.S. Template-based program verification and program synthesis. *Int J Softw Tools Technol Transfer*, 15(497–518), 2013.