Probabilistic Type Inference by Optimizing Logical and Natural Constraints*

Irene Vlassi Pandi[1], Earl T. Barr[2], Andrew D. Gordon[1,3], and Charles Sutton[1,4,5]

[1] University of Edinburgh, Edinburgh, UK
[2] University College London, London, UK
[3] Microsoft Research Cambridge, Cambridge, UK
[4] The Alan Turing Institute, London, UK
[5] Google AI, Mountain View, USA

We present a new approach to the type inference problem for dynamic languages. Our goal is to combine *logical* constraints, that is, deterministic information from a type system, with *natural* constraints, that is, uncertain statistical information about types learnt from sources like identifier names. To this end, we introduce a framework for probabilistic type inference that combines logic and learning: logical constraints on the types are extracted from the program, and deep learning is applied to predict types from surface-level code properties that are statistically associated. The foremost insight of our method is to constrain the predictions from the learning procedure to respect the logical constraints, which we achieve by relaxing the logical inference problem of type prediction into a continuous optimisation problem. We build a tool called *OptTyper* to predict missing types for TypeScript files. *OptTyper* combines a continuous interpretation of logical constraints derived by classical static analysis of TypeScript code, with natural constraints obtained from a deep learning model, which learns naming conventions for types from a large codebase. By evaluating *OptTyper*, we show that the combination of logical and natural constraints yields a large improvement in performance over either kind of information individually and achieves a 4% improvement over the state-of-the-art. The full paper of this work is available on arxiv [1].

In our view, probabilistic type inference should be considered as a constrained problem, as it makes no sense to suggest types that violate type constraints. To respect this principle, a principled framework for probabilistic type inference that couples hard, *logical* type constraints with soft constraints drawn from structural, *natural* patterns into a single optimisation problem. While, in theory, there is the option of filtering out the incorrect predictions, our framework goes beyond that; our composite optimisation serves as a communication channel between the two different sources of information.

Current type inference systems rely on one of two sources of information

(I) *Logical* Constraints on type annotations that follow from the type system. These are the constraints used by standard deterministic approaches for static type inference.

* This work was supported by Microsoft Research through its PhD Scholarship Programme.

(II) *Natural* Constraints are statistical constraints on type annotations that can be inferred from relationships between types and surface-level properties such as names and lexical context. These constraints can be learned by applying machine learning to large codebases.

Our goal is to improve the accuracy of probabilistic type inference by combining both kinds of constraints into a single analysis, unifying logic and learning into a single framework. We start with a formula that defines the logical constraints on the types of a set of identifiers in the program, and a machine learning model, such as a deep neural network, that probabilistically predicts the type of each identifier.

The key idea behind our methods is a *continuous relaxation* of the logical constraints. This means that we relax the logical formula into a continuous function by relaxing type environments to probability matrices and defining a continuous semantic interpretation of logical expressions. The relaxation has a special property, namely, that when this continuous function is maximised with respect to the relaxed type environment, we obtain a discrete type environment that satisfies the original constraints. The benefit of this relaxation is that logical constraints can now be combined with the probabilistic predictions of type assignments that are produced by machine learning methods.

More specifically, this allows us to define a continuous function over the continuous version of the type environment that sums the logical and natural constraints. And once we have a continuous function, we can optimise it: we set up an optimisation problem that returns the most natural type assignment for a program, while at the same time respecting type constraints produced by traditional type inference. Our main contributions follow:

- We introduce a general, principled framework that uses soft logic to combine logical and natural constraints for type inference, based on transforming a type inference procedure into a single numerical optimisation problem.
- We instantiate this framework in *OptTyper*, a probabilistic type inference tool for TypeScript.
- We evaluate *OptTyper* and find that combining logical and natural constraints has better performance than either alone. Further, *OptTyper* outperforms state-of-the-art systems, LambdaNet, DeepTyper and JSNice.
- We show how *OptTyper* achieves its high performance by combining *Logical* and *Natural* constraints as an optimisation problem at test time.

Fig. 1 illustrates our general framework through a simplified running example of predicting types. Our input is a minimal function with no type annotations on its parameters or result. To begin, in Box (a), we propose fresh type annotations `START` and `END` for each parameter and `ADDNUM` for the return type. We insert these annotations into the function's definition. Our *logical constraints* on these types represent knowledge obtained by a symbolic analysis of the code in the function's body. In our example, the use of a binary operation implies that the two parameter types are equal. Box (c) shows a minimal set of logical constraints that state that `addNum`'s two operands have the same type. In general, the logical

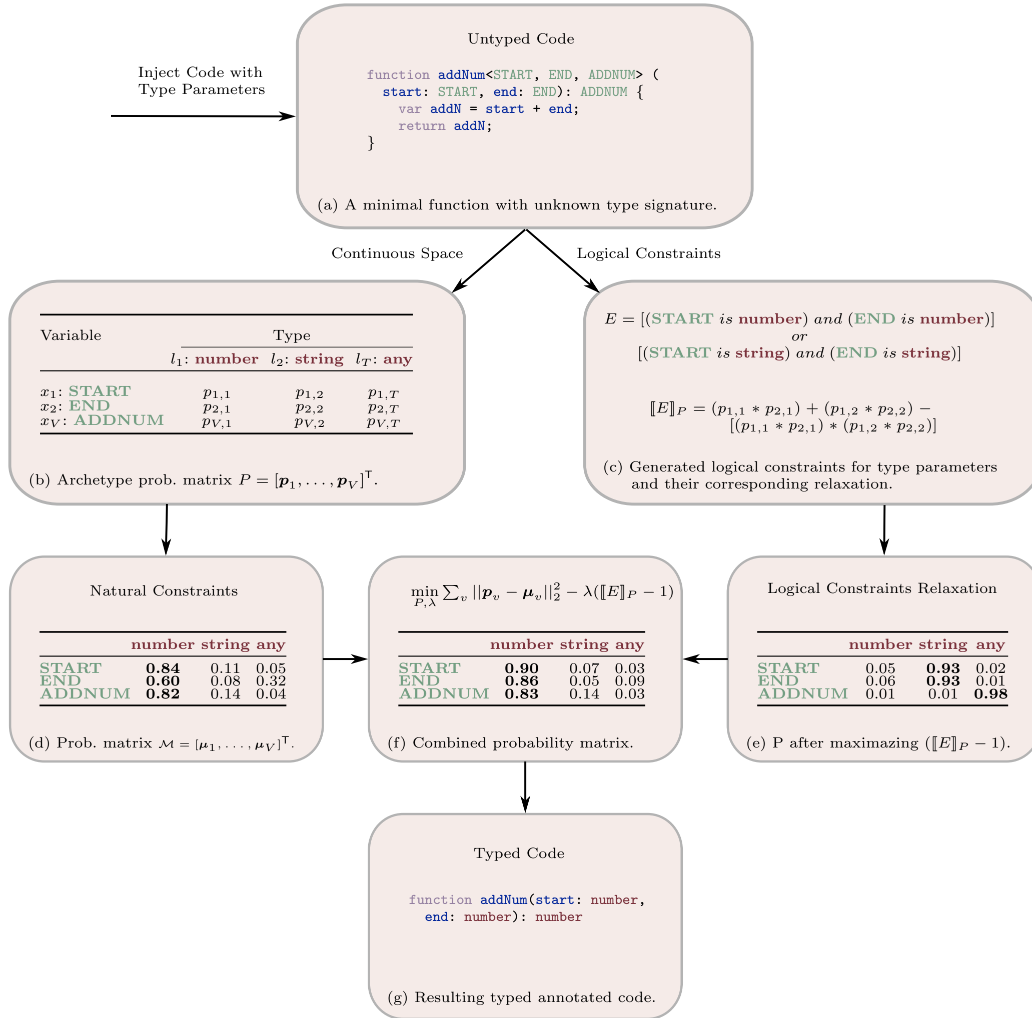

Fig. 1: An overview of the three type inference procedures via a minimal example.

constraints can be much more complex than our simple example. If we only have logical constraints, we cannot tell whether `string` or `number` is a better solution, and so may fall back to the type `any`. The crux of our approach is to take into account *natural constraints*; that is, statistical properties learnt from a source code corpus that seek to capture human intention. In particular, we use a machine learning model to capture naming conventions over types. We represent the solution space for our logical or natural constraints or their combination as a $V \times T$ matrix P of the form in Box (b): each row vector is a discrete probability distribution over our universe of $T = 3$ concrete types (`number`, `string`, and `any`) for one of our $V = 3$ identifiers. Box (d) shows the natural constraints

$\mathcal{M}$ induced by the identifier names for the parameters and the function name itself. Intuitively, Box (d) shows that a programmer is more likely to name a variable `start` or `end` if she intends to use it as a `number` than as a `string`. We can relax the boolean constraint to a numerical function on probabilities as shown in Box (c). When we numerically optimise the resulting expression, we obtain the matrix in Box (e); it predicts that both variables are strings with high probability. Although the objective function is symmetric between `string` and `number`, the solution in (e) is asymmetric because it depends on the initialisation of the optimiser. Finally, Box (f) shows an optimisation objective that combines both sources of information: E consists of the logical constraints and each probability vector μ_v (the row of $\mathcal{M}$ for v) is the natural constraint for variable v. Box (f) also shows the solution matrix and Box (g) shows the induced type annotations, now all predicted to be `number`.

References

1. Pandi, I.V., Barr, E.T., Gordon, A.D., Sutton, C.: Probabilistic type inference by optimising logical and natural constraints (2020), https://arxiv.org/abs/2004.00348

AlwaysSafe: Reinforcement Learning without Safety Constraint Violations during Training (Extended Abstract)*

Thiago D. Simão[1], Nils Jansen[2], and Matthijs T. J. Spaan[1]

[1] Delft University of Technology, The Netherlands
{t.diassimao, m.t.j.spaan}@tudelft.nl
[2] Radboud University, Nijmegen, The Netherlands
n.jansen@science.ru.nl

Publication. This is an extended abstract of [7].

Despite the astonishing successes in Reinforcement Learning (RL) [8], unsafe exploration still prevents its deployment to real-world tasks [2]. This issue has motivated the study of constrained RL to ensure safety [4]. In this framework, an agent interacts with an environment modeled as a Constrained Markov Decision Process (CMDP) [1] without knowledge about the transition, reward, and cost functions. In safe RL [6], the cost function is used as a proxy to distinguish between safe and unsafe behaviors. Therefore, the agent must find a policy with maximum expected reward among the safe policies, those with expected cost smaller than a safety threshold.

Different algorithms have been proposed for constrained RL with bounded regret in terms of performance and in terms of constraint violation, such as the `OptCMDP` algorithm [5]. However, these algorithms may still violate the constraints during the early episodes, since they encourage the agent to explore unknown parts of the environment, making their deployment to real-world tasks infeasible. We aim at developing RL algorithms that can learn without violating the constraints, that is, with no regret in terms of constraint violation.

We observe that often most of the state description is only relevant for the reward signal and does not influence the safety of the agent. In this setting, it can be easy for an expert to define the dynamics relevant for safety. Consider for instance a limit on the consecutive movements of a robot arm to avoid overheating or indicating unsafe areas such as stairs on a mobile robot's map. Such constraints may be represented in a compact model and are a prerequisite for deploying RL in practice. Hence, we assume that this compact model is known and is represented by an abstract CMDP $\bar{\mathcal{M}}$. This assumption allows the agent to explore, but **always** within the set of **safe** policies.

This work has a novel perspective on the use of abstractions. Prior work usually focuses on building a policy for $\bar{\mathcal{M}}$ that will later be executed in the

* This research is funded by the Netherlands Organisation for Scientific Research (NWO), as part of the Energy System Integration: planning, operations, and societal embedding program and the grants NWO OCENW.KLEIN.187: "Provably Correct Policies for Uncertain Partially Observable Markov Decision Processes" and NWA.1160.18.238: "PrimaVera".

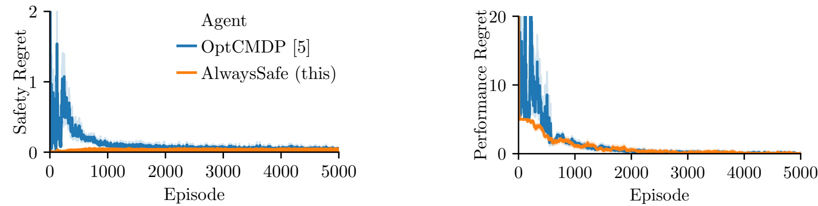

Fig. 1. Safety and performance regret of the policy executed in each episode.

ground CMDP $\mathcal{M}$. Our approach, however, focuses on computing a policy in the ground CMDP $\mathcal{M}$ and uses the abstract model $\bar{\mathcal{M}}$ to guarantee safety. This allow us to decouple the safety concerns from the reward signal.

Our contribution is four-fold: (i) we study the kind of abstraction sufficient to concisely describe and distill safety dynamics. Using factored MDPs [3], (ii) we devise an example of such abstract model. Assuming such model is given, (iii) we propose the `AlwaysSafe` algorithm, that learns an optimal policy for the CMDP without violating the constraints. Finally, (iv) we show that this algorithm has no regrets in terms of constraint violation, making it an always safe RL algorithm.

The empirical analysis (Fig. 1) showcases the capabilities of the `AlwaysSafe` algorithm: (i) it respects the constraints during training, showing no safety regret; (ii) it eventually achieves optimal performance since at the end of training it has no performance regret; and (iii) when the cost function is aligned with the reward it reduces the performance regret.

In summary, the proposed algorithm is always safe during learning, eventually reaches the optimal policy; and decouples exploration from safety issues in RL.

References

1. Altman, E.: Constrained Markov decision processes, vol. 7. CRC Press (1999)
2. Amodei, D., Olah, C., Steinhardt, J., Christiano, P.F., Schulman, J., Mané, D.: Concrete problems in AI safety (2016), arXiv:1606.06565
3. Boutilier, C., Dearden, R., Goldszmidt, M.: Exploiting Structure in Policy Construction. In: Proc. Int. Joint Conf. on Artificial Intelligence. pp. 1104–1113 (1995)
4. Dulac-Arnold, G., Mankowitz, D.J., Hester, T.: Challenges of Real-World Reinforcement Learning. In: ICML Workshop RL4RealLife (2019), arXiv:1904.12901
5. Efroni, Y., Mannor, S., Pirotta, M.: Exploration-Exploitation in Constrained MDPs. In: ICML Workshop on Theoretical Foundations of Reinforcement Learning (2020), arXiv:2003.02189
6. García, J., Fernández, F.: A Comprehensive Survey on Safe Reinforcement Learning. Journal of Machine Learning Research **16**, 1437–1480 (2015)
7. Simão, T.D., Jansen, N., Spaan, M.T.J.: AlwaysSafe: Reinforcement Learning Without Safety Constraint Violations During Training. In: Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems (2021)
8. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT press, 2 edn. (2018)

Model-based Verification of Recurrent Neural Networks for Temporal Logic Constraints

Steven Carr[1], Nils Jansen[2], and Ufuk Topcu[1]

[1] The University of Texas at Austin, USA
[2] Radboud University, Nijmegen, The Netherlands

Research in the reinforcement and supervised learning communities has demonstrated the utility of recurrent neural networks (RNNs) in synthesizing control policies in domains that exhibit temporal behavior [15,3]. The internal memory states of RNNs, such as in long short-term memory (LSTM) architectures [8], effectively account for temporal behavior by capturing the history from sequential information [12]. Furthermore, in applications that suffer from incomplete information, RNNs leverage history to act as either a state or value estimator [17] or as a control policy [7].

In safety-critical systems such as autonomous vehicles, policies that are guaranteed to prevent unsafe behavior are necessary. We seek to provide formal guarantees for policies represented by RNNs with respect to temporal logic [13] or reward specifications. Such a verification task is, in general, hard due to the complex, often non-linear, structures of RNNs [11]. Existing work directly employs satisfiability-modulo-theories (SMT) [16] or mixed-integer linear program (MILP) [1], however, such methods not only scale exponentially in the number of variables but also rely on constructions using only rectified linear units (ReLUs).

We take an iterative and model-based approach, see Fig. 1. We extract a policy from a given RNN in the form of a finite-state controller (FSC) [14]. First, we employ a modification of a discretization technique called *quantized bottleneck insertion*, introduced in [9]. Basically, the discretization facilitates a mapping of the continuous memory structure of the RNN to a pre-defined number of discrete memory states and transitions of an FSC.

However, this standalone FSC without a formal model is often not sufficient to prove meaningful properties. The proposed approach relies on the exact behavior a policy induces on a specific application that can be modeled formally. We apply the extracted FSC directly to a formal model, and the resulting restricted model is amenable for efficient verification techniques that certify whether a specification is satisfied [2].

If the specification does not hold, verification methods typically provide diagnostic information on critical parts of the model in the form of so-called counterexamples. We propose to utilize such counterexamples to identify *improvements* in the extracted FSC or in the underlying RNN. First, increasing the amount of memory states in the FSC may help to approximate the behavior of the RNN more precisely [9]. Second, the RNN may actually require further training data to induce higher-quality policies for the particular application. Existing approaches rely, for example, on loss visualization [6], but we strive to exploit the information we can gain from the concrete behavior of the RNNs with respect to a formal model. Therefore, in order to decide whether more data are needed in the training of the RNN or whether first the number of memory states in the FSC should be increased, we identify those critical decisions of the current FSC

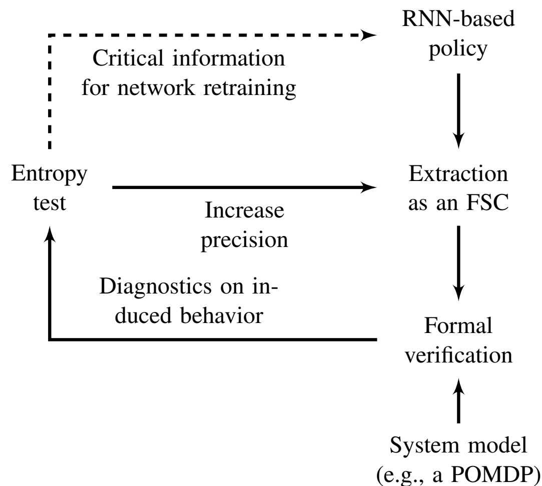

Fig. 1: High-level iterative policy extraction process.

that are "arbitrary". Basically, we measure the entropy [5] of each stochastic choice over actions according to the current FSC-based policy at critical states. That is, if the entropy is high, the decision is deemed arbitrary despite its criticality and further training is required.

We showcase the applicability of the proposed method on *partially observable Markov decision processes (POMDPs)*. With their ability to represent sequential decision-making problems under uncertainty and incomplete information, these models are of particular interest in planning and control [4]. Despite their utility as a modeling formalism and recent algorithmic advances, policy synthesis for POMDPs is hard both theoretically and practically [10]. For reasons outlined earlier, RNNs have recently emerged as efficient policy representations for POMDPs [7]. We detail the proposed approach on POMDPs and combine the scalability and flexibility of an RNN representation with the rigor of formal verification to synthesize POMDP policies that adhere to temporal logic specifications.

We demonstrate the effectiveness of the proposed synthesis approach on a set of POMDP benchmarks. These benchmarks allows for a comparison to well-known POMDP solvers, both with and without temporal logic specifications. The numerical examples show that the proposed method (1) is more scalable, by up to 3 orders of magnitude, than well-known POMDP solvers and (2) achieves higher-quality results in terms of the measure of interest than other synthesis methods that extract FSCs.

References

1. Michael E. Akintunde, Andreea Kevorchian, Alessio Lomuscio, and Edoardo Pirovano. Verification of RNN-based neural agent-environment systems. In *AAAI*, pages 6006–6013. AAAI Press, 2019.

2. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
3. Bram Bakker. Reinforcement learning with long short-term memory. In *NIPS*, pages 1475–1482. MIT Press, 2001.
4. Anthony R Cassandra. A survey of POMDP applications. In *AAAI*, volume 1724, 1998.
5. Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.
6. Ian J. Goodfellow and Oriol Vinyals. Qualitatively characterizing neural network optimization problems. In *ICLR*, 2015.
7. Matthew J. Hausknecht and Peter Stone. Deep recurrent Q-learning for partially observable MDPs. In *AAAI*, pages 29–37. AAAI Press, 2015.
8. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
9. Anurag Koul, Alan Fern, and Sam Greydanus. Learning finite state representations of recurrent policy networks. In *ICLR*, 2019.
10. Nicolas Meuleau, Kee-Eung Kim, Leslie Pack Kaelbling, and Anthony R. Cassandra. Solving POMDPs by searching the space of finite policies. In *UAI*, pages 417–426. Morgan Kaufmann, 1999.
11. Wim De Mulder, Steven Bethard, and Marie-Francine Moens. A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*, 30(1):61–98, 2015.
12. Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. In *ICLR*, 2014.
13. Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
14. Pascal Poupart and Craig Boutilier. Bounded finite state controllers. In *NIPS*, pages 823–830. MIT Press, 2003.
15. Ah Chung Tsoi and Andrew D. Back. Discrete time recurrent neural network architectures: A unifying review. *Neurocomputing*, 15:183–223, 1997.
16. Qinglong Wang, Kaixuan Zhang, Xue Liu, and C. Lee Giles. Verification of recurrent neural networks through rule extraction. *CoRR*, abs/1811.06029, 2018.
17. Daan Wierstra, Alexander Förster, Jan Peters, and Jürgen Schmidhuber. Solving deep memory POMDPs with recurrent policy gradients. In *ICANN*, pages 697–706. Springer, 2007.

Grammar Filtering For Syntax Guided Synthesis

Kairo Morton[1], William T. Hallahan[2], Elven Shum[2], Ruzica Piskac[2], and Mark Santolucito[3]

[1] MIT
[2] Yale University
[3] Barnard College

1 Extended Abstract

Program synthesis automatically derives code from a specification, which specifies what the code should do, without describing details of how the code should do it. One classic form of program synthesis is *programming by example* (PBE) [4]. In this form of program synthesis, the user only provides the synthesis tool with a set of input-output pairs, illustrating the desired behavior of the code. Then, the synthesis engine generalizes from the examples, and returns code that works on unspecified examples as well.

A variety of tools, supporting a range of input formats, have been developed to support program synthesis [5, 7, 8]. As a variety of formats leads to challenges in empirically comparing the tools, the Syntax Guided Synthesis format language was proposed [1] as a common specification language for a variety of synthesis problems, including PBE. This format has had enough success to inspire a yearly competition of SyGuS compatible synthesis tools [2].

A SyGuS problem consists of two pieces: a set of *constraints*, and a *grammar*. The constraints specify first order logic formulas which must be satisfied by the synthesized function. In the case of a PBE problem, the constraints are simply the input-output examples. The grammar is a set of functions from which the synthesized function must be constructed.

The choice of a grammar is a crucial component of a SyGuS problem. The size of the grammar is a crucial trade off. A large grammar allows for a wide range of synthesized functions, which is essential for many real world applications [9]. However, in practice, existing SyGuS solvers tend to be faster and more effective when given smaller grammars.

In this extended abstract, we outline our machine learning based technique (previously published in [6]) for automated *grammar reduction*. Grammar reduction acts as a preprocessing step to a SyGuS solver, and aims to automatically reduce the size of the grammar before the SyGuS solver is called. This way, even if a large, expressive grammar is initially specified, the SyGuS solver only considers a smaller, hopefully more relevant, subset of that grammar.

Clearly, this technique is risky: there is a danger that the grammar reduction will remove a function that is crucial for the synthesizer to succeed. Our key insight is to split the problem of filtering the grammar into two pieces: a *criticality* prediction, and a *time savings* prediction. The criticality prediction

accounts for the semantics of the problem: it predicts whether a grammatical element is needed to form a solution. The time savings prediction accounts for the implementation of a particular SyGuS solver: it predicts whether removing a grammatical element will save a solver a significant amount of time. To reduce the risk of overly filtering the grammar, we remove only those grammatical elements that we predict are both noncritical and will save the solver significant time.

Our system consists of two learned models - the criticality prediction and the time prediction. The criticality prediction uses a Feed forward neural network, while the time prediction uses an averaging of times with various grammar configurations. The most challenging part of constructing these models lies in collecting the training datasets.

The process for building a dataset for criticality prediction is informed by the type signature of the resulting neural network. Our goal in criticality prediction is to map the tuple of an input-output example and a grammar terminal to a probability that the given terminal is critical for the input-output example. To generate this set, we first use CVC4's SyGuS support [3] to generate a stream of functions in the grammar. We then generate random inputs to generate input-output examples for each function. From the function, we extract the set of grammar terminals that are present in the function. By taking all combinations of input-output examples and terminals from this set, we begin to generate our dataset. We repeat this process up to a bounded number of functions.

To build the dataset for time prediction, we run permutations of SyGuS queries with CVC4 with a different grammar element removed each time. In this way we can approximate the time saved by removing any one grammar terminal. We note that this model must be specific for the synthesis tool - our timing model built with CVC4 could not be reused for a different SyGuS solver. Currently, the time prediction model does not consider the constraints themselves in making a prediction. Instead, we simply average over our set of all SyGuS queries. Investigating optimizations to this model is an interesting open direction.

Our approach is implemented in a tool named GRT (Grammar Reduction Tool) which can be used as a black box for any existing SyGuS solver. The tool is publicly available at `https://github.com/KTMorton/Live_Programming_Research`. In our evaluation, we ran GRT on the SyGuS Competition 2019 Strings Track benchmarks. We used GRT as a front end for CVC4, the winner of that track, and found that it reduced overall synthesis times by 47.65%, reducing the time to find a solution on 32 of the 64 benchmarks. Additionally, GRT allowed CVC4 to solve one additional benchmark under the timeout limit (3600 seconds). We also ran GRT without the time prediction component of the model, and found that the system performed much worse - succeeding on only 11 out of the 64 benchmarks. This indicates that when using machine learning models for grammar reduction for program synthesis, it is important to not only consider the semantic model of the problem, but also the implementation of the synthesis tool itself.

References

1. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design. pp. 1–8. IEEE (2013)
2. Alur, R., Fisman, D., Padhi, S., Reynolds, A., Singh, R., Udupa, A.: The 6th competition on syntax-guided synthesis. `https://sygus.org/comp/2019/results-slides.pdf` (2019), accessed: 2019-11-20
3. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14, `https://doi.org/10.1007/978-3-642-22110-1_14`
4. Cypher, A., Halbert, D.C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B.A., Turransky, A. (eds.): Watch What I Do: Programming by Demonstration. MIT Press, Cambridge, MA, USA (1993)
5. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering. vol. 1, pp. 215–224. IEEE (2010)
6. Morton, K., Hallahan, W., Shum, E., Piskac, R., Santolucito, M.: Grammar filtering for syntax-guided synthesis. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34, pp. 1611–1618 (2020)
7. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. ACM SIGPLAN Notices **50**(6), 619–630 (2015)
8. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. ACM SIGPLAN Notices **51**(6), 522–538 (2016)
9. Santolucito, M., Hallahan, W.T., Piskac, R.: Live programming by example. In: Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019. (2019). https://doi.org/10.1145/3290607.3313266, `https://doi.org/10.1145/3290607.3313266`

Formal Synthesis of Lyapunov Functions and Barrier Certificates using Neural Networks

Alessandro Abate[1], Daniele Ahmed[2], Alec Edwards[1], Mirco Giaccobe[1], Andrea Peruffo[1]

[1] University of Oxford
[2] Amazon Inc

We propose an automatic and formally sound method for synthesising Lyapunov functions for the asymptotic stability, as well as Barrier certificates (or functions) for the safety analysis, of autonomous non-linear dynamics represented as systems of ordinary differential equations.

Typical methods are either analytical and require manual effort or are numerical but lack of formal soundness. Symbolic computational methods instead, give formal guarantees but are typically semi-automatic because they rely on the user to provide appropriate function templates.

We propose a method that finds Lyapunov functions or Barrier certificates automatically—using machine learning—while also providing formal guarantees—using satisfiability modulo theories (SMT). We employ a counterexample-guided approach where a numerical learner and a symbolic verifier interact to construct provably correct Lyapunov neural networks (LNNs). The learner trains a neural network that seeks to satisfy given sufficient criteria for asymptotic stability or safety over a samples set; the verifier proves via SMT-solving that the criteria are satisfied over the specified domain or augments the samples set with counterexamples.

This work is bolstered by a software tool, FOSSIL [2]. The tool has two core contributions on synthesis—automation and soundness—both of which are attained by means of the inductive, counter-example-based method. This method exploits the flexibility of candidate functions generated by training neural network templates, the formal assertions provided by the verifier, and finally new procedures to ease the exchange of information between the two mentioned components. The tool is endowed with features of usability, scalability, and robustness.

The work presentented builds on previous works involving the use of a countexmaple guided approach to synthesis of Lyapunov functions [3]; [1] then incorporates the use of neural network templates for these functions before being to extended to the synthesis of barrier certificates in [4].

References

1. Abate, A., Ahmed, D., Giacobbe, M., Peruffo, A.: Formal Synthesis of Lyapunov Neural Networks. IEEE Control Systems Letters **5**(3), 773–778 (2021)

2. Abate, A., Ahmed, D., Edwards, A., Giacobbe, M., Peruffo, A.: Fossil: A software tool for the formal synthesis of lyapunov functions and barrier certificates using neural networks. In: HSCC (2021, In Print)
3. Ahmed, D., Peruffo, A., Abate, A.: Automated and Sound Synthesis of Lyapunov Functions with SMT Solvers. In: TACAS (1). LNCS, vol. 12078, pp. 97–114. Springer (2020)
4. Ahmed, D., Peruffo, A., Abate, A.: Automated formal synthesis of neural barrier certificates for dynamical models. In: TACAS. LNCS, vol. 12651, pp. 370–388. Springer (2021)