Department of Electrical, Computer, and Software Engineering

Part IV Research Project

Literature Review and

Statement of Research Intent

Project Number: 77

Automatic Program Generation Research Intent

Buster Major

Nathan Cairns

Jing Sun, Gill Dobbie

28/10/2019

Declaration of Originality

This report is my own unaided work and was not copied from nor written in collaboration with any other person.

Name: Buster Major

ABSTRACT: The problem of automatically generating a program from some specification is well-studied and has a long history. Generally viewed as either a learning or search problem, there are competing approaches, a select few of which we aim to develop by building upon. Trends and technologies are discussed, as well as our research intent.

1. Literature Review

The problem of generating a program automatically from some specification has long been a topic of research in within communities of computer science research, albeit with different approaches. For example, program generation is viewed as a problem of enumerating and evaluating candidate programs from a search space by the programming languages (PL) community, whereas the machine learning (ML) community views it as a problem of learning to map specifications to programs with training [1].

1.1. Program generation by learning

The learning approach to automatic program generation typically harnesses the power of recurrent neural networks (RNNs) to translate a set of input / output (I/O) examples to a functional program [2, 3]. The choice of using I/O examples as a program specification is popular because it alleviates the need for a programmer to write a formal program description to use as input to the program generation model. This is preferable because writing formal program specifications if often more difficult than writing a program alone [2]. Within the machine learning community there are two predominant competing approaches to the problem of *automatic program learning*; (1) neural program synthesis, and (2) neural program induction. In addition to these approaches, there is also the emergence of the use of learning translation rules using some mapping or machine translation framework [4, 5].

1.1.1. Neural program synthesis

The exercise of mapping some program specification to a program is fairly well studied in research, and the current trend is to use RNNs for this kind of learning. Delvin *et al.*, Singh *et al.*, Balog *et al.*, Sutskever *et al.* and Ling *et al.* all employ similar two stage neural architectures for training; one network for transforming a variable length specification into a constant length latent vector (encoder) and a further network to map this specification vector to a program (decoder) [2, 6-9]. This approach is needed because neural networks find it hard to deal with variable length input vectors, so some size normalization should be done [6, 8]. While all researches mentioned employ neural architectures for this it is of note that Sing *et al.* rely on a reverse-recursive-reverse neural network (R3NN) because of their need to incrementally generate program trees within their domain specific language (DSL) [6], while Balog *et al.* use simple feed forward architectures for their ease of training and (in their case) performance empirically equivalent to that of an RNN [7]. Delvin *et al.* and Ling *et al.* use conventional RNNs in their encoding and decoding processes [2, 9]. It is important to note that this two-

tiered architecture is modular in its design, and its pointed out the extent to which the vector to program (decoder) network is decoupled from the specification to vector (encoder) network, meaning once trained the decoder could in theory be used in conjunction with a different encoder, hence different specification type [6]. This is important because it allows a large amount of flexibility with our representation if we choose to head in this direction and means we can freely change our specification choice without having to retrain/remodel.

1.1.2. Neural program induction

Neural program induction works by inducing a *latent* (non-observable) representation of a program through learning processes. While still strongly based in machine learning, it provides a different view of program generation because the generated program is hidden within the learned model. For example, Devlin *et al.* train a neural architecture to generate outputs from inputs, the outputs of which should correspond to a correct program implementation given the input [2]. While relevant, this approach is not overly useful for us as a goal of our research is to generate a full program.

1.1.3. Reinforcement learning / Statistical Machine Translation

In contrast to the trend of using neural architectures in learning program generation problems, there is also the approach of learning a mapping between representation and program through some probabilistic reinforcement learning framework. Reinforcement learning is a way of learning which harnesses the notion of some reward mechanism, implemented here by some value function. Branavan *et al*. implement this function as a validation function, checking whether the learned mapping results in the correct *action* for some set of instructions. Their approach focuses on the transformation of natural language (Windows troubleshooting instructions) to executable actions [4]. A similar approach to Branavan *et al*. (albeit without using action-based reinforcement learning), Oda *et al.* propose a framework for generating pseudo-code from existing code snippets using a technology called statistical machine translation, a framework designed for learning mappings between languages. Statistical machine translation was originally intended for translation between natural languages like English and Japanese, and in this research it is extended to Python code and corresponding pseudo-code [5]. Regarding our research this can be thought of as performing the inverse of generating a program from a representation, and because the approach in this paper transforms a relatively concrete language into a more abstract representation, looking to do the inverse using similar technologies may prove challenging. Furthermore, although this approach is not strictly program generation it provides important insight into the technologies existing in the program generation domain, and it gives an example of program specification that is not in the form of I/O examples. Unlike Branavan *et al*.'s approach of using the more generalizable reinforcement learning concept, this approach seems limited to the specification domain of languages (which we may not want to limit ourselves to) due to the use of the statistical machine translation tools.

Similarly, Xu *et al*., while not strictly using a statistical machine translation tool also harness the potential of learning transformation rules between languages. Their research studies the potential of learning a set of mapping rules from a *formal* specification to MATLAB code, and while differently framed from Oda *et al.*'s research it still studies the transformation of a concrete specification to a more or less equivalently concrete output (algebra to MATLAB) [10], which we do not feel is overly transferrable to our research as we are looking to use a more abstract specification format.

1.1.4. Existing technologies

Gaunt *et al.* has designed a probabilistic programming language for specifying inductive program synthesis problem called *TerpreT*. The programmer provides a program representation in the form of a partial program which *TerpreT* maps to inputs and outputs, thereby *inferring* the program [11]. Although similar in many ways to program *sketching* [12], this language is tailored towards inductive program synthesis problems and also borrows from machine learning techniques like *gradient descent* and model training. This program representation the system learns is *latent*.

1.2. Program generation by search

While the ML approach to program generation is an exercise in mapping specification features to some output program through supervised training with examples, generating a program through searching requires searching through a (possibly infinite) program search space and checking candidate programs through some validation mechanism [6]. Some aspects of ML are borrowed in this approach. Popular trends in representation type include partial programs and I/O examples [3, 12, 13]. Because it is desired to output programs which are dynamic and expressive in language, program search spaces can become very large very quickly, so there are several ways of reducing the effective search space including enumerative, stochastic, constraint based and version space algebra-based techniques [6]. There is also a well-established set of existing technologies which implement program synthesis through search.

1.2.1. Searching

Large search spaces are the main drawback of the searching approach, and it is negated in several ways by various researchers. Alur *et al.* use a version space algebra to drastically reduce their search space by defining the program specification as a partially implemented program. They discuss how (in their case) the synthesis is guided by the syntax of the partially implemented program, the writer of which can assert *holes* in the functionality left up to the synthesizer to search for and implement. The specification also includes I/O examples used to verify the correctness of the implementation [13]. This notion of a partial implementation or a *hypothesis* is also explored by Singh *et al.*, who use ML technologies (as discussed in 1.1.1) to build a model that learns how to search. While still distinctly a search problem, this approach works by training the two-tiered neural architecture discussed in 1.1.1. Their second stage neural

architecture can either learn to search over the program space, where the network can be asked to either optimize over syntactic similarity to training data (i.e. the traditional ML approach) or learn a controller that selects different partial program trees and incrementally expands them (the conventional search approach). The former of these requires no form of program validation as it is not an exercise in searching, and although embedding continuous representations of semantics is challenging, optimizing over syntactic similarity removes the need for this all together. A downside of using this technique is it removes potentially semantically equivalent programs from the search space which may also be valid solutions, causing the synthesis to be very narrow minded [6].

1.2.2. Inductive synthesis

Indictive synthesis is the approach of generalizing a program from specification in the form of multiple examples, typically in the form of I/O. Because it learns from examples it can be considered a further form of machine learning, but it retains the searching approach at its core. Inductive synthesis comes in 2 forms, first being *active learning*, which uses the notion of querying one or more *oracles* to generate both positive *and* negative training examples to validate candidate programs. The other form known as *counterexample guided inductive synthesis* (CEGIS) also uses an oracle to generate examples and these are used to validate candidate programs found from searching. With CEGIS, if a candidate program is falsified with an example φ generated from the oracle, φ is returned to the learning algorithm and the search is repeated [13, 14]. Oracles are typically implemented in a technology called a Satisfiability Modulo Theory (SMT) (not to be confused with *statistical machine translation* discussed earlier) constraint-based solver, which is essentially a back-end engine used for model checking using first order logic expression of a theory T [15].

1.2.3. In conjunction with machine learning

While searching a learning can be considered alternative approaches to program generation they are frequently combined in research. As mentioned in 1.1.1, Sutskever *et al.* employ the typical ML two stage neural architecture, albeit a more specialized form called a *Long Short-Term Memory* (LSTM) architecture, a form of neural architecture which deals well with long range temporal dependencies that are common in normalizing specification length [8]. They use a further technique called *beam search* which is a technique used to search through partial hypotheses or partially complete programs and extend these incomplete outputs to all possible complete states. It then assigns the partial output a probability of correctness, thereby providing means to prune the search space by discarding the most unlikely hypotheses [8]. Other researchers have also used beam search to good levels of success [16], with Devlin *et al.* noting they believe it has helped their generation accuracy by providing a wider stochastic hypothesis space [2].

1.2.4. Existing technologies that generate programs through search

Perhaps most significant existing (search based) technology is the Microsoft Excel 2013 feature *FlashFill* which allows users to define examples of string manipulations/transformations by defining I/O, the program then generates a regex like program which satisfies the users specifications. It is worth mentioning that because the model only generates output for the user, *FlashFill* is agnostic to generating a program or generating output straight away from a latent program representation [2, 3]. The result of Singh *et al.*'s research is a tool called *SKETCH* which is able to fill in holes of a partially implemented program. A programmer is able to write code declaratively and let the synthesizer find a suitable implementation by searching, the process of which also takes advantage of SMT solver technologies [12]. λ^2 is a further technology which is able to generate functional programs which manipulate/transform recursive data structures, the implementation of which combines generalization, deduction and enumerative searching [3, 7]. This tool is of interest because it can generalize I/O examples into partial programs, similar to what is done in SKETCH and may be useful to model similar specification transformations on in our research. With respect to SMT solvers, there is an open source toolkit called *Moses* that which provides complete out-of-the-box solution for constraint-based solutions and translations [17].

2. Research Intent

We intend to answer the question *can a program be automatically generated from a specification other than I/O using machine learning techniques.* We believe this question is worth answering because this domain has the potential to have a large impact on software development practices moving forward. Verified design models built with machine learning generating operational software could potentially streamline software development methodologies by reducing quality/testing overhead, reducing cost of development and lowering factors such as human error. While it is clear program generation from some specification is a well-studied and understood problem, my partner and I have identified several gaps in the research which we look to explore and potentially develop. We have decided to peruse a machine learning approach to our research due to the overhead of restricting such a search space, and the lack of learning ability in a lot of the search techniques. It also appears to us that choosing a more lightweight framework for exploring our research will likely be more achievable. The two-tiered architectures discussed in 1.1.1 and 1.2.3 also promise to be robust, well understood/established and most importantly extensible. It is for these reasons we will look to emulate an architecture similar to the two-tiered RNN architecture, meaning we will be able to treat our training and representation type separately during the development and training period. The usefulness of LSTMs discussed by Sutskever *et al.* could also prove to be a useful RNN type to use for normalizing specification lengths for encoding. There is also a trend where

each research paper aims to solve program generation for a very specific domain, for example Windows troubleshooting instructions [4], real-time algorithmic software [18] or recursive data structure transformation functions [3]. It is likely we will choose a domain as restrictive as these in order to ensure this research is achievable. Our representation will likely not be in the form if I/O examples as we recognize this is a well understood approach in the domain and we believe there is little *achievable* research we could produce using this in conjunction with the two-tiered architecture. With regards to representation, it is important that we choose a format that is not too formal to avoid the overhead that Balog *et al.* encountered with having to formulate a non-trivial DSL, while being formal enough to avoid the issues we foresee with using pseudocode like Oda *et al.* as discussed in 1.1.3. As a result, we have concerns over using overly informal specifications like natural languages (e.g. pseudo-code). We noticed a trend in using somewhat declarative languages as representations, which may also be worth using as they map well to machine learning practices [6]. Figure 1 shows our proposed high-level design for this project. It visualizes the components that make up the implementation and allow us to break down our project into units of work. As discussed, we will decide on a specification by evaluating several candidates by abstraction level. We will then define a set of programmatic rules which transform the specification into our DSL. We anticipate we will be able to easily define these because we will be choosing a target DSL with a restricted domain, so declarative statements should not have much ambiguity when mapping to specific statements within the DSL.

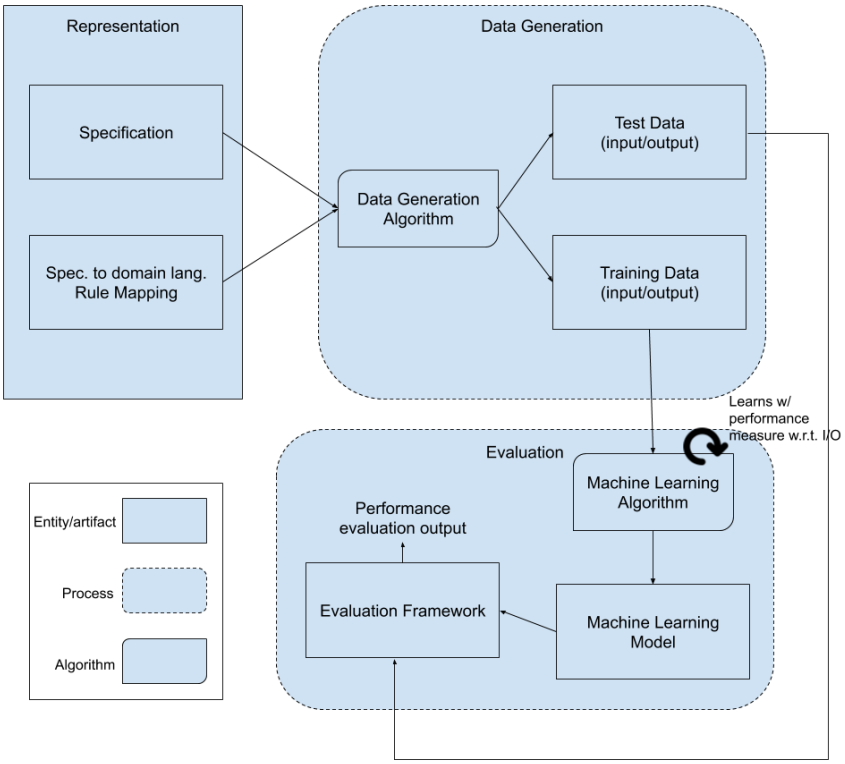

Figure 1 - Proposed implementation design

Our DSL with be in the language of Python and restricted to a certain domain/function type (e.g. data structure transformations, string manipulations, programmatic arithmetic etc.). Using this specification and rule set we will be able to generate a set of synthetic data for use on our machine learning model, one set for training and one set for evaluation. We will evaluate the choice of machine learning algorithm and model by experimenting with several different models, fine tuning parameters and conducting further research on potential architectures. It is likely our models will be a form of the two-tiered neural architecture discussed in 1.1.1 and 1.1.4, and for reasons discussed in 2. Finally, to evaluate the success of our model we will find or build an evaluation framework to give us quantitative results on the quality of our model. We will conduct further research on appropriate technologies for this. Figure 2 shows our work breakdown plan in the form of a Gantt chart.

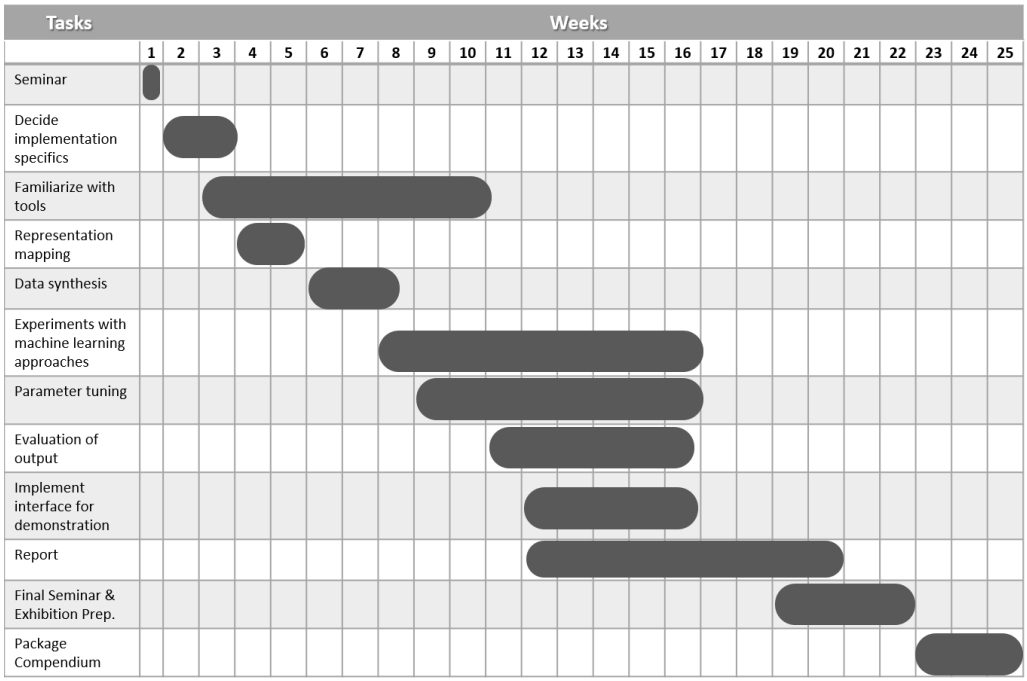

Figure 2 – 25-week project plan starting Monday 6 May 2019

3. Conclusions

While the domain of automatic program synthesis is well-studied, there are opportunities to build upon the current research by combing and building on existing approaches. The intent of my partner and I is to follow a two-tiered neural architecture to generate programs from a restricted functional domain while using a specification format distinct from I/O examples, while synthesizing our own training data from a set of pre-defined transformation rules. We have created a 25-week plan which outlines goals and the timeline we need to follow in order to achieve success with this project.

References

[1] R. Simmons-Edler, A. Miltner and S. Seung, "Program Synthesis Through Reinforcement Learning Guided Tree Search," Jun 7,. 2018.

[2] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed and P. Kohli, "RobustFill: Neural Program Learning under Noisy I/O," Mar 21,. 2017.

[3] J.K. Feser, S. Chaudhuri and I. Dillig, "Synthesizing data structure transformations from input-output examples," ACM SIGPLAN Notices, vol. 50, pp. 229-239, Jun 3,. 2015.

[4] S.R. Branavan, H. Chen, L.S. Zettlemoyer and R. Barzilay, "Reinforcement Learning for Mapping Instructions to Actions," Aug 1,. 2009.

[5] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda and S. Nakamura, "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T)," pp. 574-584, Nov 2015.

[6] R. Singh and P. Kohli, "AP: Artificial Programming," Jan 1,. 2017.

[7] M. Balog, A.L. Gaunt, M. Brockschmidt, S. Nowozin and D. Tarlow, "DeepCoder: Learning to Write Programs," Nov 7,. 2016.

[8] I. Sutskever, O. Vinyals and Q.V. Le, "Sequence to Sequence Learning with Neural Networks," Sep 10,. 2014.

[9] W. Ling, E. Grefenstette, K.M. Hermann, T. Kočiský, A. Senior, F. Wang and P. Blunsom, "Latent Predictor Networks for Code Generation," Mar 22,. 2016.

[10] J.Y. Xu and Y. Wang, "Towards a Methodology for RTPA-MATLAB Code Generation Based on Machine Learning Rules," pp. 117-123, Jul 2018.

[11] A.L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor and D. Tarlow, "TerpreT: A Probabilistic Programming Language for Program Induction," Aug 15,. 2016.

[12] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia and V. Saraswat, "Combinatorial sketching for finite programs," ACM SIGARCH Computer Architecture News, vol. 34, pp. 404, Oct 20,. 2006.

[13] R. Alur, R. Bodik, G. Juniwal, M.M.K. Martin, M. Raghothaman, S.A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak and A. Udupa, "Syntax-guided synthesis," pp. 1-8, Oct 2013.

[14] S. Gulwani, "Synthesis from Examples: Interaction Models and Algorithms," pp. 8-14, Sep 2012.

[15] C. Barrett and C. Tinelli, "Satisfiability Modulo Theories," in Handbook of Model Checking, E.M. Clarke, T.A. Henzinger, H. Veith and R. Bloem, Cham: Springer International Publishing, 2018, pp. 305-343.

[16] S. Minton and S.R. Wolfe, "Using machine learning to synthesize search programs," pp. 31-38, 1994.

[17] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin and E. Herbst, "Moses: Open Source Toolkit for Statistical Machine Translation," Jun 1,. 2007.

[18] T.E. Smith and D.E. Setliff, "Towards an automatic synthesis system for real-time software," pp. 34-42, 1991.