MSc Artificial Intelligence
Master Thesis

Type-supervising Neural Program Synthesis using Typed Holes

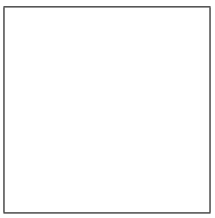
by
Tycho Grouwstra
6195180

December 12, 2019

48
2019-2020

Supervisor:
MSc. Emile van Krieken

Assessor:
Dr. Annette ten Teije



IVI

1 Research Direction

Program synthesis is the task to automatically construct a program that satisfies a given high-level specification, be it a formal specification, a natural language description, or input-output examples.

This differs from program induction, commonly known as 'supervised learning', in that fitting the model into the discrete form of a given grammar forces one to instantiate such a model from probabilistic data interpretations, whereas in traditional supervised learning, such probabilistic interpretations of the data may be taken at prediction time. This means that in program synthesis, we are able to distill our modeled function to a simplified discrete form that may well be intelligible to humans as well, opening up opportunities for human-machine cooperation in writing software. And in fact, in a novel functional programming paradigm called *type-driven development* [1] based on typed holes, this is already happening.

While considered a holy grail of computer science, program synthesis is a challenging task, characterized by large search spaces, e.g. a space of 10^{5943} programs discovering an expert implementation of the MD5 hash function. [2]

Program synthesis was traditionally studied as a computer science problem, but has recently been explored using machine learning approaches under the name of neural program synthesis [5]. However, there has not yet been much work combining these two approaches.

Whereas the traditional approaches in program synthesis focused on constraining the large discrete search space, neural program synthesis instead tends to incrementally generate programs, using continuous representations of the state space to predict the next token, be it in a sequential fashion [7], or in a structured one based on abstract syntax trees (ASTs) [6].

While the typically incremental approach of neural program synthesis breaks down the problem by considering one part of the program at a time, the original challenge remains.

Whereas a program synthesizer may be programmed or taught to output programs adhering to a given grammar, there is typically no guarantee that such partial program constructions will qualify as a full executable program adherent to the grammar. As a result, neural synthesizers will have little intermediary feedback to go by, limiting their effectiveness.

But if only complete programs can be evaluated for validity and behavior, then it is much harder to provide synthesizers with an accurate understanding of incomplete programs.

We hypothesize that the effectiveness of neural program synthesis may be improved by adding type information as additional features during training.

Research question: can neural program synthesis methods benefit from using types as additional features?

2 Expected Contribution

The present work aims to be the first experiment to:

- bring the type-based information traditionally used in program synthesis into the newer branch of neural program synthesis;
- bridge the two fields, such as to find a best-of-both-worlds golden mean;
- use this type info to better constrain the search space, improving the effectiveness of (neural) program synthesis methods.

3 Existing work

4 Experiment

User intent in program synthesis can be expressed in various forms, including logical specification [**?**], examples, traces, natural language, partial programs, or even related programs. [2] Such logical specifications might also include types of inputs and outputs, and this branch of program synthesis has commonly focused on using functional programming languages as the synthesis language. [**?**] [**?**] [**?**] [**?**] [**?**] [**?**] For our purposes here, we will focus on synthesis based on input-output examples, aka programming by example (PBE), as having known input-output types will help provide us the needed type information we would like to use.

As we are interested in types, we will need to build upon AST-based (or node-based) rather than sequential (or token-based) synthesis methods. 1 Type-based approaches to synthesis are based on *deductive search*, a top-down search strategy. Such type-based deductive search is most powerful in a setting where the underlying

[1] `https://manning.com/books/type-driven-development-with-idris`

DSL is loosely-constrained, that is, permits arbitrary type-safe combinations of subexpressions. In particular, any ML-like calculus with algebraic datatypes can serve as a core language for type-driven synthesis. [2]

Particularly useful for our purposes is a programming language feature called *typed holes* [3], as used in Agda [2], Idris [3], and Haskell [4]. Typed holes allow one to write an incomplete functional program template, enabling type inference for any such holes, in turn enabling suggested completions to the user. Such interactive program synthesis has been labeled as the solver-aided programming method *sketching* [2] or *type-driven development*. As such, one of our goals is to improve suggested hole completions in such languages to facilitate this interactive programming style.

To get the most out of our types, we will want to provide them for:

- inputs
- outputs
- abstract syntax trees (ASTs)

The basic idea here is simple: our program should return the desired type, while taking the desired input types.

However, in the simple case, the implications of this info do not require much computation: one will just restrict their search to those AST nodes allowed for the present hole as dictated by the grammar.

What makes the use of types more powerful in restricting the search space is the use of *parametric polymorphism*: a function *append* may work using either lists of numbers or lists of strings. [5] As such, its type signature may be made *generic* such as to have its return type reflect the types of the parameters used. Having such information available at the type level may add additional information over what is used in the simpler case above.

It should be noted however that not all programs benefit from this, but only those for which potential building blocks include statically typed functions with parametric return types.

As a result, this restriction limits our method from being used for direct synthesis of programming languages that are dynamically typed, e.g. Ruby language, as well as to those lacking parametric polymorphism, e.g. C language. However, this only partly hinders those wishing to synthesize programs in such languages: rather than synthesizing in such languages directly, we would suggest synthesizing from the language that best constrains the search space, then using source-to-source translation methods (e.g. neural machine translation [4]) using the synthesized program as input to obtain a program translated into the target language.

In other words, languages with stronger type systems are preferable for program synthesis. This is well-known in the traditional program synthesis community, where various researchers have been making use of the Haskell language, a statically typed, purely functional programming language with type inference and lazy evaluation. [6]

The advantages of Haskell as a synthesis language over the above-mentioned ML lies in its focus on the functional paradigm, unlike e.g. OCaml: this paradigm is more amenable to static types, which for synthesis helps us constrain our search space.

As a benchmark algorithm we will use [6], a top-down incremental neural synthesis method which like other neural methods is not presently making use of type-level information.

We had trouble finding suitable benchmark task in the literature.

On the one hand, benchmark tasks within non-neural program synthesis methods have usually consisted of only a limited number of tasks [**?**] [1] [**?**] [**?**] [**?**] [**?**], as such non-neural methods do not require an equivalent to the training set typically used in machine learning setups.

On the other hand, benchmark tasks within Neural Program Synthesis have often remained somewhat simpler in nature, rendering them less fit for our purposes — either being imperative in nature (and as such being less amenable to types), such as sorting tasks [8] [7], or otherwise focusing on a simple set of types — such as strings [6] —or differently put, not including building blocks with parametric return types.

As a third issue, it was also common for existing papers to keep their datasets unpublished [6] [**?**] [**?**].

As a result, it looks like we would need to create a benchmark task, or if anything a training dataset compatible with one of the existing benchmark tasks, ourselves.

We will do this by taking a library of basic operations, and generating any possible functions using combinations thereof within a given complexity threshold. Whereas modern programming languages might have a broad plethora of grammatical constructs available, for the purpose of our proof-of-concept we will opt to hide much of this. We will do this by taking a queue from the lambda calculus, which opts to view constructs as functions, both so as to provide a unified view of various operations, as well as to enable *currying* of functions. A curried

[2] `https://agda.readthedocs.io/en/latest/getting-started/quick-guide.html`

[3] `http://docs.idris-lang.org/en/latest/tutorial/typesfuns.html#holes`

[4] `https://wiki.haskell.org/GHC/Typed_holes`

[5] Left out of scope here are *refinement types*, which further aid synthesis based on conditions.

[6] `https://www.haskell.org/`

version of a function is not unlike its original form, yet allowing arguments to the function to be applied to it one at a time. The way this works is that, when an argument is applied to a curried form of a function taking two parameters, the result is a function that still takes one parameter, before yielding the actual result of the original function. As such, viewing traditional operators such as addition as curried functions can both simplify the way we view things, while also increasing expressiveness. And in fact, this is indeed what happens in various functional languages, where traditional operators such as `+` are viewed simply as infix operator forms of curried functions. In Haskell language, for example, '+' may be used to refer to the addition operator in infix notation, whereas '(+)' may be used to refer to its form as a traditional function.

As such, we might express as curried functions not only basic mathematical operators, but by wrapping them as library functions, also traditional grammatical constructs as (functional) conditionals and function composition operators. Grammar-wise, this should leave us with few options: referencing variables (whether from our library or parameters), and creating lambda functions.

The reason we consider programs of a tree-based form, rather than as a list of imperative statements such as variable definitions, is that the view of programs as function compositions guarantees us that any complete program will yield us output of the desired type. This guarantees us that, rather than just branching out, our search will focus on finding acceptable solutions.

To simplify the problem, we would disallow the use of any constant values such as free-style strings.

The generation process is as follows:

- we take our function, containing a number of parameters, a return type, and a body consisting of a hole;
- we enumerate the variables we have that could (if applicable, after function application) return a type compatible with our desired return type, to create versions of the function with said hole plugged with these variables (which in the case of functions, may, in turn, contain holes);
- we filter these potential programs given their number of remaining holes based on our complexity threshold;
- we add the remaining programs to a queue of complete or incomplete programs, depending on whether they still contain holes;
- we similarly process the functions in the queue of incomplete programs, until we are left with an enumeration of complete programs;
- we evaluate the complete programs by the input-output examples, to filter down to those matching our behavioral criteria;
- finally, we generate sample inputs for each program, and calculate their corresponding outputs.

5 Result

Having added our type-level supervision during training, we expect synthesis success rates to rise and required evaluations to drop compared to the baseline algorithm. This demonstrates that the findings from traditional program synthesis methods are relevant also in the field of neural program synthesis.

References

[1] Feser, J. K., Chaudhuri, S., and Dillig, I. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 229–239.

[2] Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends in Programming Languages 4*, 1-2 (2017), 1–119.

[3] Hashimoto, M., and Ohori, A. A typed context calculus. In *Theoretical Computer Science* (1997), Citeseer.

[4] Kalchbrenner, N., and Blunsom, P. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (2013), pp. 1700–1709.

[5] Kant, N. Recent advances in neural program synthesis. *CoRR abs/1802.02353* (2018).

[6] Parisotto, E., Mohamed, A., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *CoRR abs/1611.01855* (2016).

[7] Pierrot, T., Ligner, G., Reed, S. E., Sigaud, O., Perrin, N., Laterre, A., Kas, D., Beguir, K., and de Freitas, N. Learning compositional neural programs with recursive tree search and planning. *CoRR abs/1905.12941* (2019).

[8] Reed, S., and De Freitas, N. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279* (2015).