Analysis of Inductive Program Synthesis Techniques on General Program Synthesis Tasks

Edward Pantridge
MassMutual Financial Group
Amherst, Massachusetts, USA
epantridge@massmutal.com

Thomas Helmuth
Washington and Lee University
Lexington, Virginia
helmutht@wlu.edu

Nicholas Freitag McPhee
University of Minnesota, Morris
Morris, Minnesota 56267
mcphee@morris.umn.edu

Lee Spector
Hampshire College
Amherst, Massachusetts, USA
lspector@hampshire.edu

ABSTRACT

A variety of inductive program synthesis (IPS) techniques have emerged from a spread of research fields in recent decades. However, these techniques have not been adequately compared on general program synthesis tasks. In this paper, several state-of-the-art methods of inductive program synthesis are compared across a set of problems involving the search for programs dealing with various data types, control structures and number of outputs. The problem set also spans tasks that are generally approached using languages at the level of machine code, assembly language, and higher level languages. Although there are many helpful metrics to consider when comparing these methods, this comparison is mainly focused on success rate. The systems investigated in this paper include: Flash Fill, Magic Haskeller, TerpreT, and Genetic Programming. After comparing the solution rates of these methods on the problem set, the results suggest that Genetic Programming and, to an extent, TerpreT are more capable of finding solutions than Flash Fill and Magic Haskeller.

Keywords

CCS CONCEPTS

•Software and its engineering → Genetic programming; •Theory of computation → *Evolutionary algorithms;*

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference format:
Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2017. Analysis of Inductive Program Synthesis Techniques on General Program Synthesis Tasks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017, Berlin, Germany, July 15–19, 2017 (GECCO '17),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

GECCO '17, Berlin, Germany

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Since the creation of Inductive Program Synthesis (IPS) in the 1970s[4], researchers have been striving to create systems capable of generating programs competitively with human intelligence. Modern IPS methods often trace their roots to the fields of machine learning, logic programming, evolutionary computation and others.

The similarities and differences of these methods have been discussed[4], but their performance is rarely compared on problem sets that could provide concrete insight into the capabilities and limitations of each method.

A demonstrative problem set has been compiled that assess an IPS method's ability to work within a range of levels of abstraction[1], manipulate a variety of data types, produce complex control structures and produce an arbitrary number of outputs of various forms[3].

This investigation is exclusively considering a methods ability to find solutions. Other measures, such as runtime or hardware models, are not discussed. In order to asses if a method can find solutions to a problem, the problem must first be phrased, in its entirety, to the method. This is not always possible.

The conclusions drawn from this comparison will speak to the flexibility of each considered method, as well as each method's success rate.

2 CURRENT STATE OF THE ART

2.1 Flash Fill

Flash Fill is a program synthesis technique found in Microsoft Excel [2]. It was designed to help non-programmers perform repetitive tasks that would otherwise require them to write Excel macro programs. It specializes in tasks that require string manipulations.

To test Flash Fills performance with our problem set, an Excel spreadsheet with one column per input and one column for output was created for each problem. Each spreadsheet included training data, which had both the input and output column populated, and unseen testing data, which left the output column cells empty.

Flash Fill is deterministic and analytic, thus it was only run once per problem on a single data set.

Excel does not include a native vector or list data structure, and it is not clear what the best way to phrase problems that require vectors to Flash Fill. A string representation of vectors was attempted for some problems, as well as putting each item from each vector

```
((f [1, 2, 3] == [0, 1, 2]) && (f [-1, 1] == [-2, 0]))
```

Figure 1: An example predicate that can be supplied to MagicHaskeller. This particular predicate produces a solution to the Decrement problem.

in a separate cell. These approaches occasionally yielded results, but it is not clear what the optimal usage is.

It is not possible to pose tasks to Flash Fill that required multiple outputs. This includes problems that require printing values in addition to returning an output. Asking Flash Fill to generate each output value in separate cell was considered, but this would be two separate tasks and would not be comparable to other the other IPS methods.

Flash Fill does not support vector types. If a problem specifies an input value will be a vector of a fixed length, the problem can be posed with each element of the vector in its own cell. If the vectors length is not fixed, this cannot be done, because a tabular structure cannot be formed. If a problem requires a vector output, it cannot be posed to Flash Fill with elements in their own cell because generating each output value would be a separate IPS task.

Due to these shortcomings of Flash Fill, there are a number of problems in this comparison that Flash Fill was unable to attempt. This is discussed more in section 4.

2.2 MagicHaskeller

MagicHaskeller is a web based that synthesizes functions in the Haskell programming language based on example function calls and their desired output in the form of a predicate. To do this, MagicHaskeller generates a stream of functions that have the same signature as the example function calls. This includes: the same number of inputs, the same input types, the same number of outputs and the same output types.

Generated functions are tested against the input predicate, and a sample of passing functions are presented to the user. The user can then "Exemplify" the suggested solution functions to see how they would behave given a variety of other inputs. If the user cannot find a solution function, more function can be generated until the entire stream has been processed.

This process relies heavily on memoization. Being a web hosted service, all MagicHaskeller users share the same dynamic programming table, and the users cannot select which functions are included when searching.

Figure 1 shows an example input predicate. Notice that multiple nested predicates can be supplied. The number of nested predicates that can be given to MagicHaskeller is very limited, and if too many predicates are given the system will produce memory errors. This is a consequence of MagicHaskeller being a web hosted service where resources are shared between all users. Due to this limitation, it was not possible to give MagicHaskeller the same training dataset as the other IPS methods. Although this weakens the presented results, it speaks to the usability and flexibility of the MagicHaskeller system.

MagicHaskeller supports the common primitive data types such as: integers, floats, strings and characters. Vectors and tuples are also supported, witch greatly expands the number of problems that were able to be posed to MagicHaskeller.

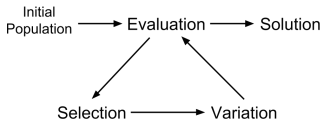

Figure 2: Evolutionary Computation process.

MagicHaskeller only allows for the synthesis of functions that produce a single output value. In order to pose question that require multiple outputs, MagicHaskeller was given predicates that specified a tuple as an output. For problems that specify that values should be printed in addition to the output value, it was considered sufficient if MagicHaskeller could synthesis a program that produced a single string containing all printed values (including whitespace and newline) as an element of its output tuple.

Due to MagicHaskeller being hosted on the web, it is difficult to embed MagicHaskeller in other systems.

2.3 TerpreT

TerpreT is a recently developed, probabilistic programming language that is designed for inductive program synthesis. Problems are specified in the TerpreT language, which is then translated into four different back-end inference algorithms: Forward Marginals Gradient Descent (FMGD), Integer Linear Programming (ILP), Satisfiability modulo theories (SMT) and SKETCH.

The TerpreT system attempts to solve IPS problems using these back-end algorithms and returns source code containing the successful parameters found by the successful back-end algorithm, if one is present.

Note that there is currently no publicly available implementation of TerpreT and thus only results on benchmark problems provided by the original authors could be obtained. It would be extremely valuable to compare TerpreT's performance on the rest of the problem set using in this paper once an implementation becomes available.

2.4 Genetic Programming

Soon after the rise of evolutionary computations, John R. Koza recognized that evolution could be used for more than optimizing a fixed structure of values. In the 1990s, Koza built upon genetic algorithms in such a way that produced executable programs. This technique, named Genetic Programming (GP), is considered inductive program synthesis because it uses input-output examples (refered to as test cases in the field of GP) to evolve a function. In fact, IPS was one of the origional motivations for creating the field of GP[5].

Genetic Programming works by generating a initial population of random programs. Traditionally these programs are represented as trees where non-leaf nodes each denote a function. The children of each non-leaf node are used as the arguments to their parents. Leaf nodes denote terminal values that could be either a constant or input value. This initial population of random programs then follows the cycle in Figure 2 until a solution is found or the run is considered a failure.

2.4.1 PushGP. For this comparison, PushGP was selected as the Gennetic Programming method. PushGP evolves programs in a Turing complete, stack based language called Push. Push features separate stacks for each data type, including code. Push programs are lists of instructions and literals. Literals are values that get placed on the stack corrisponding to their type. Instructions are built-in functions that pop values off the stacks, modify them, and push them back on the appropriate stacks. Programs are run through an interpreter, which modifies the stacks. After all instructions and literals have been processed through the interpreter, the final state of the stacks is the output of the program [6].

Why PushGP for IPS?

Implementations of PushGP systems are available in the Clojure programming language[1], as well as a new implementation in python[2].

3 PROBLEMS

3.1 Basic Execution Models Problems

The first set of problems were taken from [1] and were intended to demonstrate TerpreT's ability to synthesis programs in a variety of execution models that span multiple levels of abstraction. These execution models are: Turing Machine, Boolean Circuits, Basic Block, and Assembly Language.

As stated by [1], the problems in this set progress from more abstract execution models towards models which resemble assembly languages. This makes these problems demonstrative of how a system performs across a variety of low-level domains.

The problems in this set are describes below:

3.1.1 Invert. Given a binary string (binary tape), invert all the bits.

3.1.2 Prepend Zero. Insert a 0 in the first index of a binary string and shift all other bits to the right.

3.1.3 Binary Decrement. Given an input binary string equal to a positive decimal number, return a binary string equal to the input number decremented by one.

3.1.4 2-bit Controlled Shift Register. Given input bit (r_1, r_2, r_3), return the same bits, except swap the order of r_2 and r_1 if $r_1 == 1$.

3.1.5 Full Adder. Given a carry bit, c_{in}, and two argument bits, a and b , output a sum bit, s, and carry bit, c_{out}, such that $s+2c_{out} = c_{in} + a_1 + b_1$.

3.1.6 2-bit Adder. Given input bits a_1, a_2, b_1, and b_2, output s_1, s_2, and c_{out} such that $s_1 + 2s_2 + 4c_{out} = a_1 + b_1 + 2(a_2 + b_2)$.

3.1.7 Access. Given an input array, V, and a positive integer, i, return V_i. Assume $0 < i < |A|$.

3.1.8 Decrement. Given an input array, V return a new array, U such that $U_i = V_i - 1$.

[1]https://github.com/lspector/Clojush
[2]https://github.com/erp12/Pysh

	TerpreT	Flash Fill	MH	PushGP
Invert	✓	x	✓	✓
Prepend Zero	✓	✓	✓	✓
Binary Decrement	✓	x	x	x
2BCSR	✓	x	x	✓
Full Adder	✓	x	x	✓
2 Bit Adder	✓	x	x	x
Access	✓	x	✓	✓
Decrement	✓	x	✓	✓

Figure 3: Results of all 4 systems on the Basic Execution Models problems from [1]. A check denotes the system could find a solution. An x denotes the problem was fully posed to the system, but a solution was not found. All problems were able to be fully posed to each system.

3.2 General Program Synthesis Benchmark Suite

The second set of problems used in this comparison was proposed by [3] in order to provide the field of Genetic Programming with a set of non-trivial benchmark problems.

This problems set included problems that deal with multiple data types, including strings, numbers, boolean and vectors. There are also multiple problems in the set that require multiple output values, or printing certain values to the screen.

Given that the Basic Execution Models problem set is designed to test a systems ability to perform in low-level domains (binary circuits, assembly language, etc), the General Program Synthesis Benchmark Suite is an excellent addition, given that the problems' origins assume a much higher-level implementation (ie. java).

4 RESULTS

Which problems were not able to be posed to certain systems

Which systems performed better on which problems

5 CONCLUSION

Limitations of the comparisons.

What does this mean for applications of IPS.

6 THINGS TO MENTION SOMEWHERE

- Differences in runtime
- FashFill and MagicHaskeller come up with same thing every time. GP might not.
- Can you extract the solution program? Is it readable?
- We used boolean vectors as binary tape from TerpreT problems.
- We considered the Access problem and the List-k problem to be synonymous in most contexts.

7 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grants No. 1617087, 1129139 and 1331283. Any opinions, findings, and conclusions or recommendations expressed

	Flash Fill	MH	PushGP
Number IO	x	✓	✓
Small Or Large	x	x	✓
For Loop Index		x	✓
Compare String Lengths	?	x	✓
Double Letters	?	x	✓
Collatz Numbers	?	x	x
Replace Space with Newline		x	✓
String Differences		x	x
Even Squares		x	✓
Wallis Pi	?	x	x
String Lengths Backwards		✓	✓
Last Index of Zero	?	x	✓
Vector Average	?	✓	✓
Count Odds	?	x	✓
Mirror Image	?	x	✓
Super Anagrams	?	x	✓
Sum of Squares	?	x	✓
Vectors Summed	?	✓	✓
X-Word Lines		x	✓
Pig Latin	?	x	✓
Negative To Zero		✓	✓
Scrabble Score	?	x	✓
Word Stats		x	x
Checksum	?	x	x
Digits		x	✓
Grade	?	x	✓
Median	?	x	✓
Smallest	?	✓	✓
Syllables	?	x	✓

Figure 4: Results of all 3 systems on the Software Synthesis Benchmark Suite from [3]. A check denotes the system could find a solution. An x denotes the problem was fully posed to the system, but a solution was not found. No symbol denotes the problem could not be fully posed to the system.

1014538503543

in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. TerpreT: A Probabilistic Programming Language for Program Induction. *CoRR* abs/1608.04428 (2016). http://arxiv.org/abs/1608.04428

[2] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. *SIGPLAN Not.* 46, 1 (Jan. 2011), 317–330. DOI: http://dx.doi.org/10.1145/1925844.1926423

[3] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. ACM, New York, NY, USA, 1039–1046. DOI: http://dx.doi.org/10.1145/2739480.2754769

[4] Emanuel Kitzelmann. 2009. Inductive Programming A Survey of Program Synthesis Techniques. (2009).

[5] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

[6] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (March 2002), 7–40. DOI: http://dx.doi.org/doi:10.1023/A: