Recursive Program Synthesis in Python

Re-implementing ESCHER in Python

JONATHAN WANG, Rutgers University

RAHUL TRIVEDI, Rutgers University

1 Introduction

Program synthesis via input-output examples opens the door for inexperienced programmers to be able to synthesize programs through desired results. Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid develop an inductive synthesis algorithm titled ESCHER that interacts with users via input-output examples and is parameterized by a set of components allowed to appear in the program. The algorithm goes a step further by being able to synthesize recursive programs. ESCHER uses two phases in order to generate programs and provide efficient synthesis by combining programs via conditionals. ESCHER uniquely uses its own data structures to increase efficiency compared to normal SAT based synthesis tools. In this paper, we reimplement this technique for use via Python. With its simple syntax, Python allows users with little experience to be able to pick up coding easily.

ESCHER's goal is to generate a recursive program that matches the given input output. Programs are generated in a bottom up fashion. Consists of branching if-else statements with constricting conditions. Users provide a list of input output examples, the set of components they want to restrict the program by, and the function they want to produce to be used for recursive evaluation.

As the original paper implemented the algorithm in OCaml, a very niche programming language, our goal is to recreate ESCHER in Python, a more widely used and easy to pick up language. We follow the steps in the paper for our algorithm and try to come up with a fully working implementation. Our algorithm mostly follows their implementation as a blueprint. However, since the paper leaves logical blank spaces for us to fill, we make our own alterations to the algorithm as well. Then, we measure the benchmarks of our performance and build upon the existing work by exploring alternate heuristic methods and benchmarking the readability of generated programs.

Ultimately, we were able to come up with a near fully working implementation aside from some bugs that we were unable to fix within the time scope of this project. Going forward, the goal is to further explore heuristic optimization and, of course, iron out the bugs.

2 Related Work

The main inspiration for this project is the published paper by Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid: *Recursive Program Synthesis* via The University Toronto and Microsoft Research. This paper is the original development of ESCHER and details the algorithm flow and implemented structures. We follow the steps in this paper and test our implementation according to their base components and test cases.

3 Methodology

ESCHER is an algorithm developed with the purpose of generating recursive programs to match a given input output and provided function referred to as the *Oracle*. It has access to a set of base components and a self call that it uses to synthesize new recursive functions. Programs are generated in a bottom up fashion consisting of if else statements with constricting conditions that string together programs generated from the base components. For example, given input outputs and a length function by the oracle, ESCHER would aim to generate a recursive function:

length(x) := if(isEmpty(x)) then zero() else inc(length(tail(x))

ESCHER is divided into two phases, the Forward Search Phase and the Conditional Inference phase. The algorithm alternates between the two phases based on some guiding heuristic function for programs. In the Forward Search Phase, ESCHER grows the set of programs by applying components to existing synthesized programs in the search space and filters out equivalent terms. In the Conditional Inference Phase, ESCHER uses the programs found in the Forward Search phase and makes use of a new data structure called the goal graph to combine those programs with if-else statements to try and generate a satisfying program. The goal graph takes the place of the traditional if, then, and else statement and detects when two programs can be joined by a conditional statement. The goal graph is able to generate conditionals for programs that satisfy a subset of the target goal, replacing the traditional if-then statements. Each level alternates between these two phases and the space of programs and resolvers increase each level.

3.1 Forward Search Phase

3.1.1 Heuristic Function

The heuristic function is the guiding principle behind ESCHER. The heuristic matches generated programs to some value measure of its cost. In generating programs, ESCHER aims to minimize the cost of programs and thus minimize the heuristic value. When traversing levels of search space, the algorithm generates programs based on some mapping of level to heuristic i.e. level 1 → generate programs of heuristic value 1. In addition, when eliminating equivalent programs, that is programs that generate the same outputs on all inputs, the algorithm prioritizes keeping the equivalent program with the smallest heuristic cost.

The most basic heuristic, and the base heuristic function implemented in our code, is simply a mapping of the size of a program.

In our implementation, ESCHER runs an infinite loop that switches between the two phases and, at each iteration, it increases the traversed level. In the forward search phase, this level is used to map the appropriate heuristic value programs to generate. At level 1, the program only generates programs of heuristic value 1, level 2 generates programs of heuristic value 2, and so on until it finds a program that resolves the input outputs and is recursively sound. When eliminating equivalent programs, we store the lowest heuristic value function and remove the rest.

3.1.2 The Oracle

The Oracle is a provided function that generates appropriate outputs given a set of inputs. That is, it is the original function that ESCHER is trying to synthesize into a recursive function. Since programs are generated in a bottom up fashion, there is no way to call the generated self call function when recursing. In order to evaluate self calls, the Oracle is instead called. This helps test recursion by assuming that the program is always correct recursively. This leads to some error cases, i.e. the program isn't actually correct in all cases. To handle this, the saturation rule is implemented (more information below).

For ESCHER, the Oracle is a necessary step. However, many works that build upon ESCHER attempt to remove the need of an Oracle since having to provide a base function greatly hinders the use cases of the algorithm.

For our implementation, the Oracle is implemented on a test by test basis. Each test has its own matching function and that function is simply declared within each test and passed to Escher as an argument. Then, when evaluating Self nodes, the Oracle function is called on the provided arguments. The only workaround is that every Oracle function can only take one argument, an array of actual arguments. This choice was made to handle the fact that different Self functions can have varying numbers of arguments.

3.1.3 Termination and Saturation

In generating valid recursive programs, it is necessary to ensure that those programs always terminate. However, since programs rely on the Oracle for self evaluation there is no way to check whether programs actually terminate. To handle this, the terminate and saturate rules are implemented. Saturate generates new inputs from the current inputs and further tests programs on those generated inputs to ensure correctness. Terminate ensures that each program is not non-terminating by examining the conditions of the recursive call.

For our implementation, saturate and terminate are used in tandem to measure a recursive program's validity. Saturate takes a program and then recursively applies the programs stored as arguments in the self call on the input parameters in order to generate a new input. Then it tests on those new inputs and repeats recursively. Here is an example flow of saturate:

Input([1,2,3]), Program: length:= if(isEmpty(x)) then zero() else inc(length(tail(x))
Saturate Tests [1,2,3] on program: Correct → Recursive call length(tail(x)) → new_input = tail([1,2,3])
→ Saturate on new Input → repeat recursively until hit a terminating case.

Our terminate rule is essentially built into the saturate rule. When saturating, it resources until it hits a terminating condition. In the case that it doesn't hit a terminating condition within a provided threshold of recursive levels, it will return that the program is non-terminating and, as a result, is invalid. Otherwise, if it does hit a terminating condition, the terminate rule is satisfied.

3.1.4 Searching

The search space consists of all possible generated programs from the base components. Searching consists of growing the search space to cover higher cost/heuristic value programs and then removing any equivalent programs where the outputs generated for all inputs are the same. At each level, new programs should be added to the search space which can then be used as components in the next level of searching.

For our implementation, searching is represented by two functions, one for growing the search space and one for eliminating equivalent values. Grow simply loops over the existing search space and applies the base component functions to all possible values within the search space. Then it checks these programs for their heuristic value and, if the heuristic is within the allowed limit, appends the new program to the search space. Eliminating equivalents just goes over all programs in the search space and maps them to their output vector on the supplied inputs. If their output vector is already within the mapping, then it compares heuristic values of the current program with the previous program and, if the current heuristic is less than the previous, the current program replaces the previous program mapping to the output vector.

3.2 Conditional Inference Phase

3.2.1 Goal Graph

The goal graph is a bipartite graph and consists of goals, resolvers, and edges. Goals are output vectors that are used as checkmarks to meet in order to synthesize a satisfying program. Resolvers are branches that satisfy a goal and contain the if, then and else subgoals necessary for satisfying a parent goal. Edges hold the connections between goals and resolvers, with the resolvers on the left indicating that the resolver is for a goal and with goals of the left indicating the goal is a subgoal needed to be met for the resolver.

The goal graph was represented using a class GoalGraph(). The goal graph has 4 fields, a root representing the output vector from the input/outputs, a list R representing the resolvers, a list G representing the goals, and a list E representing the edges between goals and resolvers.

Resolvers are represented using a class Resolver(). The resolver has 6 fields, the 3 lists ifgoal, thengoal, and elsegoal represent the goal vectors for the if, then and else conditions and 3 programs ifSat, thenSat and elseSat are fields for programs that satisfy those conditions.

3.2.2 Splitting Goals

The splitgoal rule is responsible for creating branches off existing goals by finding programs that satisfy a subset of the required outputs of the parent goal. In splitgoal(), the function takes in the list of synthesized programs, the goal graph, and a list of inputs and outputs. For each synthesized program, their output is based on the list of inputs and is compared against each goal in the graph to see if there are any partial matches. If a partial match is found, there is first a check to see that there is no other resolver with the same if then condition. If there is no such resolver, a new resolver is created with the boolean vector produced by the program as the if condition and the program as the then condition's satisfied program. The else vector of outputs is produced as the opposite of the then vector of outputs. These goals are connected to the new resolver and are also inserted as subgoals in the goal graph. Edges are added between the resolver and the subgoals into the graph, as well as the resolver to its parent goal.

3.2.3 Resolving

The resolve rule is responsible for matching programs to goals in resolvers and finding a satisfiable program for the root goal. In resolve(), for each resolver in the graph, programs are checked against the if, then and else vectors to find any matches. If a match is found, they are linked to the resolver. There is then a check to see if all the fields are full in which case the goal is complete. There is a recursive call here to resolving() to check for if satisfying this goal has completed other goals as well. If the goal that was satisfied is the root goal, then the program is returned.

4 Code/Results

Attached to this document is our full workspace for our implementation. In it, you will find the following programs:

Program Name	**Purpose**
forward_search.py	Contains logic for the Forward Search phase of ESCHER. Handles initializing search space, growing search space, heuristic calculations, and filtering out equivalent programs. *Logic mentioned above.*
escher.py	Contains logic for the Conditional Inference phase of ESCHER. Handles Goal Graph structure, splitting goals into subgoals, and resolving with a working program. Also contains the main ESCHERfunction which handles the overall flow of the algorithm. *Logic mentioned above.*
structures.py	Contains all AST node definitions. Sets up the logic for all base components and evaluations. Also handles the SATURATE case and recursive testing. *More information below.*
escherNestedIfElse.py	Same as escher.py but attempts to handle nested if elses. *More information below.*
escherTesting.py	File containing multiple test cases for different recursive functions. *More information below.*
escherTestingNestedIfElse.py	Same as escherTesting.py but attempts to handle nested if elses. *More information below.*

One minor issue with our initial code was that our algorithm was only able to synthesize recursive programs that could be resolved with a single if else statement. The way we implemented Conditional Inference didn't quite work on cases with nested if statements. To try and resolve this we tried a completely new implementation of conditional inference but were unable to fully iron out the bugs.
As a result, we included escher.py, which contains the fully working implementation on only single if else programs, and escherNestedIfElse.py, which contains the partially working implementation on both single and branching if else programs. There are also escherTesting.py and escherTestingNestedIfElse.py, which

are just the respective testing programs for each implementation. Below we will show the benchmarks and developed programs for different test cases on both of these programs.

When searching through possible programs, it is crucial that the algorithm is provided with a set of base components that is sufficient in synthesizing a working program. These base components differ from problem to problem and the complexity/time of the algorithm scales exponentially with the amount of possible base components. Because of this, we wanted to provide a minimal set of base components that would allow us to synthesize programs for our test cases. Structures.py contains all the logic for these base components:

Returns integer	INTVAR, NUM, PLUS, MINUS, TIMES, INCNUM, DECNUM, NEG, DIV2, ZERO, HEAD
Returns boolean	FALSE_exp, TRUE_exp, AND, OR, NOT, EQUAL, ISEMPTY, ISNEGATIVE, ISPOSITIVE, LT
Returns list	LISTVAR, TAIL, CONS, CONCAT, INCLIST, DECLIST, EMPTYLIST, ZEROLIST

Using these base components, we tested our code on the following functions and logged the resultant programs:

Note: Tests were run with a minimal set of base components in the search space in order to decrease complexity and run time. If all components were included, the run time would be much higher.

Function	Time(s) *Escher.py*	Time(s) *EscherNestedIfElse.py*	Generated Program *Escher.py*	Generated Program *EscherNestedIfElse.py*

length()	0.003	0.004	if (isEmpty(x))
			then 0
			else
			incrementnum(length(tail(x))
				if (isEmpty(x))
				then 0
				else
				incrementnum(length(tail(x))

reverse()	0.458	0.460	if (isEmpty(x))
			then []
			Else
			concat(reverse(tail(x)), cons(head(x), []))
				if (isEmpty(x))
				then []
				Else
				concat(reverse(tail(x)), cons(head(x), []))

square_list()	0.102	0.0836	if ((-x) >= 0)
			then []
			else
			concat(square_list(decrementnum(x)), cons((x*x),[]))
				if ((-x) >= 0)
				then []
				else
				concat(square_list(decrementnum(x)), cons((x*x),[]))

stutter()	0.0414	0.175	if (isEmpty(x)) then x	if (isEmpty(x)) then x

			else
			cons(head(x),cons(head(x),stutter(tail(x)))
				else
				cons(head(x),cons(head(x),stutter(tail(x)))

insert()	9.961	11.985	if ((-y) >= 0)
			then (cons(z,x))
			else
			insert_list([(insert_list([(tail(x)), (decrementnum(y)), z,])), (-y), (head(x)),])
				if (isEmpty(x))
				then (cons(z,x))
				else
				(insert_list([(insert_list([(tail(x)), (decrementnum(y)), z,])), (-(decrementnum(y))), (head(x)),]))

fib()	3.28	3.23	if ((x == 0)||(x == 1))
			then x
			else
			fib(decrementnum(x))+fib(decrementnum(decrementnum(x)))
				if ((x == 0)||(x == 1))
				then x
				else
				fib(decrementnum(x))+fib(decrementnum(decrementnum(x)))

sum_under()	0.197	*N/A*	if ((-x) >= 0)
			then 0
			else
			x+sum_under(decrementnum(x))
				N/A

compress()	*N/A*	0.911	*N/A*	If (isEmpty(x))
				then x
				else
				    if(head(x) == head(tail(x)))
				    Then compress(tail(x))
				    Else
				   cons(head(x), compress(tail(x))

last_in_list()	*N/A*	0.002	*N/A*	If (isEmpty(x))
				Then head(x) #Returns error
				else
				    if(isEmpty(tail(x))
				    Then head(x)
				    Else
				    last_in_list(tail(x)

drop_last()	*N/A*	0.007	*N/A*	If (isEmpty(x))
				Then x
				Else
				    If (isEmpty(tail(x))
				    Then []
				    Else
				   cons(head(x),
				drop_last(tail(x)))

evens()	*N/A*	0.028	*N/A*	If (isEmpty(x))
				Then x
				Else
				    If (isEmpty(tail(x))
				    Then x
				    Else
				 cons(head(x),
				evens(tail(tail(x))))

contains()	*N/A*	*N/A*	*N/A*	*N/A*

Note:
length() is length of list
reverse() is reversed list
square_list() returns a list of running squares up to the square of an integer, i.e square_list(3) = [1,4,9]
stutter() stutters elements in a list, i.e. stutter([1,2,3]) = [1,1,2,2,3,3]
insert() inserts into a list
fib() returns fibonacci numbers
sum_under() returns the running summation up to a number, i.e sum_under(4) = 1+2+3+4
compress() compresses subsequent duplicates, i.e. compress([1,1,2,2]) = [1,2]
last_in_list() gets last item in a list
drop_last() removes the last item from a list
evens() returns a list of only the even indexes of a list
contains() returns whether an integer is in a list

As you can see, escher.py works in synthesizing any program that can be synthesized in one if else. Meanwhile, escherNestedIfElse.py works for most examples of one if else and most examples of nested if else. Ideally going forward, we will continue to bugfix escherNestedIfElse.py in order to get it working on all examples. In addition, the measured time for most of the examples is very fast. To the human eye, most of them were near instant. The ones that went longer have to do with increasing complexity of programs and the inefficiencies of the grow algorithm. After each level the search space grows exponentially, meaning the higher the level required (which determines the heuristics of programs and their complexity), the higher the complexity and run time.

5 Novelty/Lessons

For the novelty factor and building on the implementation, we decided to look at heuristic functions and the readability of generated programs. The heuristic function is what guides the searching algorithm and, as a result, determines the efficiency of the algorithm in finding programs. Additionally, the readability of generated programs is extremely important as, if the programs aren't readable, we won't be able to discern the actual meaning.
For heuristic functions, we experimented with a few possibilities. The base of our heuristic function was the size of the program. For example, program head(x) would have heuristic 1 and would be generated at level 1, meanwhile program head(tail(x)) would have heuristic 2 and would be generated at level 2. This is the simplest heuristic function and we experimented mostly with this since it was all we could manage

with the scope of our project. We identified how minor changes would drastically affect the runtime in cases. For example, programs with multiple inputs like or, and, and plus would be generated much less than single input examples. This was because the size of the program would scale with the size of both inputs. To combat this, we had multiple input sizes be set to the maximum size of all its inputs plus one. This led to a much more even distribution of components in our search space and allowed us to find working programs at much earlier levels. Additionally, since we are generating recursive functions, the Self call is incredibly important. In order to increase the priority at which self call programs are generated, we initialized self calls to be one less than standard library calls. For example, self(tail(x)) would have heuristic 1 while head(tail(x)) would have heuristic 2. As a result, we found that recursive programs would generate much faster meaning that the possible programs would get generated earlier, thus decreasing the level of complexity and runtime. Going forward, we could test different measures of determining a program's importance. For example, we could use machine learning to match programs with an importance value based on previous examples.

For human readability, the benchmarks are much more subjective. An inexperienced programmer may see results as completely unreadable while programmers would see them as readable. In seeing whether the results are readable or not, feel free to look at the generated programs above. In our opinions, most of the generated programs are very readable and simple. There are some cases where they are strangely complex, but these may be due to a multitude of factors like the heuristic function not fitting the example well or the test cases not being sufficient enough to cover all cases. In the test cases where we were sure the inputs were minimally sufficient, the generated programs were very simple. Generally, the programs are generated as if conditions matching the base cases of a recursive function and then a recursive call on a decreasing factor of the input, which is the standard formula for a recursive function. Ultimately, we determined that the algorithm did indeed generate human readable programs. But, considering we are very familiar with the algorithm and recursive programming, the results may not be human readable for inexperienced programmers.

5 Conclusion

Working towards the future, efficiency can greatly be approved through various fixes across the program. A potential area to explore is improving the way we look through the graph for matches. Currently, searching through edges and resolvers is incredibly inefficient as it loops through every entry in the list. Some potential solutions to this include making a structure to represent goals that can hold edges to resolvers and subgoals for ease of access. This will greatly reduce the time it takes to iterate through lists across splitgoal() and resolve().

Additionally, more work on generating programs with nested conditionals can increase the space of programs that can be generated by ESCHER in python. Currently, the program is held back by inefficiency in generating conditionals, causing some programs to time out. This can be fixed by exploring more efficient ways of generating conditionals such as looking into creating resolvers on partial matches for if conditions.

The work done on ESCHER in python has been promising with many programs being generated in a matter of milliseconds. As we continue to develop the algorithm, we hope to be able to improve the speed while increasing the functionality and ability to produce more complex programs.

6 References

Albarghouthi, Aws, et al. *Recursive Program Synthesis*. University of Toronto, Microsoft, 2016, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/cav13.pdf.