**Function synthesis from code traces**

George B. Herson

College of Computing, Georgia Institute of Technology, Atlanta, Georgia, USA, gherson3@gatech.edu

ABSTRACT

This research investigates development of a proof of concept named Exemplar[1] that broadens where PbE can be applied by requiring unstructured code traces showing the processing from exemplary input to output. Deduction and search are involved, as can interrogation of the user. The resulting functions come with a suite of useful unit tests, and are relatively easy to polish into finished function.

BACKGROUND

Once a programmer decides what a program should do, programming becomes the discovery of the precise mechanism that will behave as desired under all legal inputs. Because that is much harder than simply specifying desired output given the exemplary inputs for a limited number of use cases, as in unit test coverage, “programming by example” (PbE) systems have long been sought. Unfortunately, success has been limited to tightly circumscribed application areas, such as resolving points of programmer uncertainty in an algorithm by search[2], or deriving spreadsheet formulae by deduction[3].

At the same time, it is common for coders to complete code traces on paper as part of the design phase before a significant programming implementation.

Therefore, it would facilitate programming if these two elements, viz., traced use cases along with their starting input and ultimate output (unit tests), would suffice for the automatic derivation of general purpose functions. The presently described system seeks to provide such a foundation.

DISCUSSION

Exemplar’s central innovation is to attack the core problem PbE systems face, i.e., the disambiguation of a desired algorithm from input and output example pairs, by introducing two compromises, (code trace) assertions and, where necessary, straightforward questions put to the user.

How Exemplar works, to the user

Exemplar frees the coder from having to initially think about how to solve a task generally and instead solve the problem in the particular, without thought to algorithmic structure beyond its linear order of operation.

Exemplar’s function derivations also usually include multiple choice questions put to the user. Such questions appear wherever necessary to confirm which of the possible generalizations are desired. When answered correctly, a given question is only asked once, and never by the synthesized funtion.

Next are two in-depth demonstrations of Exemplar.

Imagine that we want to generate a primality function with only the following “trace”. (Text after # signs are comments.)

# Leading less-than and greater-than signs denote input and output, respectively, to the function to be built.

<1008 # Assertion-less user examples like this lead to a unit test but are  
>False # ignored re: function synthesis.  
  
<0  
i1==inp # Rename variable from the default, "i1". (The "i1==" is optional.)

inp <= 1 # I.e., because inp is <=1, we:  
>False # return False.  
  
<1  
i1==inp   
inp <= 1 # Ditto.  
>False  
  
<3   
>True  
  
<4  
i1==inp

inp > 1 # Because inp is greater than 1, we continue with:  
j == 2 # FOR loop with only one iteration in this example because:  
inp % j == 0 # inp divides evenly. And so we have an answer:  
>False # not prime.  
  
<5  
i1==inp  
inp > 1 # Since inp is > 1, the following (j) loop is reached.  
j==2  
# (N.B. inp % j != 0, <4 example’s condition NOTed, is \*not\* traced here because

# that truth does not trigger any code.)  
j==3 # j is simply incremented, until it  
j==4  
j==5  
j==inp # reaches the value of inp, at which point we conclude that inp is:  
>True # prime.

To decipher the above, Exemplar attempts to wind up the loop iterations that the user’s examples imply into a self-consistent ball of Python code, phrased as a function and embedded in a unit test suite.

In this case, file TestPrimeNumber.py is created with contents

*# AUTOGENERATED FILE -- RENAME OR YOUR EDITS WILL BE OVERWRITTEN***import** unittest, exemplar  
actual\_io\_trace = **''** *# Receives test values print()'ed and input().*global\_input = [] *# Assigned in each test to provide input() values to the function under test.  
  
  
# print() is mocked to see if the tests recreate the .exem-specified i/o in actual\_io\_trace.***def** print(line=**""**) -> **None**:  
 **global** actual\_io\_trace  
 **if** line **is** str:  
 line = line.translate(str.maketrans({**"'"**: **r"\'"**})) *# Escape single quotes* actual\_io\_trace += **">"** + str(line) + **'\n'***# input() is mocked to return the test-specified input as well as add it to actual\_io\_trace.***def** input(variable\_name: str = **""**) -> str:  
 *# (variable\_name is ignored because it may not have been specified by the .exem.)* **global** actual\_io\_trace  
 result = global\_input.pop(0)  
 result = result.translate(str.maketrans({**"'"**: **r"\'"**})) *# Escape single quotes* actual\_io\_trace += **"<"** + result + **'\n'** *# Eg, '<Albert\n'* **return** result  
  
  
*# The generated function under Stage 2 (i.e., a test per example) testing.***def** prime\_number():  
 inp = int(input(**"inp:"**)) *# Eg, 0* **if** inp<=1:  
 print(**'False'**)  
 **return 'False'   
 elif** inp>1:  
 **for** j **in** range(2, 6, 1):  
 **if** j==inp:  
 print(**'True'**)  
 **elif** inp%j==0:  
 print(**'False'**)  
 **break  
  
  
class** TestPrimeNumber(unittest.TestCase):  
  
 **def** setUp(self):  
 **global** actual\_io\_trace  
 actual\_io\_trace = **''** self.maxDiff = **None  
  
 def** test\_prime\_number3(self):  
 **global** global\_input  
 global\_input = [**'1008'**] *# From the .exem* prime\_number() *# The function under test is used to write to actual\_io\_trace.* self.assertEqual(**'''<1008  
>False  
'''**, actual\_io\_trace)  
  
 **def** test\_prime\_number8(self):  
 **global** global\_input  
 global\_input = [**'0'**] *# From the .exem* prime\_number() *# The function under test is used to write to actual\_io\_trace.* self.assertEqual(**'''<0  
>False  
'''**, actual\_io\_trace)  
  
[This continues, one test per user example.]  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 unittest.main()

That generated function again:

1. **def** prime\_number():
2. inp = int(input(**"inp:"**)) *# Eg, 0*
3. **if** inp<=1:
4. print(**'False'**)
5. **return 'False'**
6. **elif** inp>1:
7. **for** j **in** range(2, 6, 1):
8. **if** j==inp:
9. print(**'True'**)
10. **elif** inp%j==0:
11. print(**'False'**)
12. **break**

How and why Exemplar created each line:

1. The function’s name, prime\_number, is taken from the input filename, prime\_number.exem.
2. All the user examples started with an integer input, so we assume the target function starts by accepting an integer of input.
3. Each example has one of two IF assertions in this position. This assertion Exemplar saw first, so it is coded into Python above its sibling IF branch.
4. Both examples with line 3’s IF assertion follow it with output of False. (There’d be an inconsistency error if not.)
5. return is implied by the extent of the conforming examples.
6. This IF condition, being mutually exclusive with the first, becomes an ELIF.
7. Each of the examples with line 6’s IF condition are followed with a FOR loop that ranges up to 6 – 1.
8. Those FOR loop instances start with IF conditions of j==inp or inp%j==0. In the case of the former,
9. True is output.
10. In the user examples with this IF condition,
11. False is output and the
12. FOR loop ends early, implying this break.

N.B. A shortcoming in this rendition of prime\_number() is that the FOR loop’s encoded range() matches that of the longest user example rather than being a WHILE loop that iterates as long as it takes to determine primality. This limitation cannot be overcome until Exemplar implements WHILE loops, a planned improvement.

Here is another problem Exemplar solves, a child’s guessing game [4]. A code trace that works for this is

# User wins.  
>Hello! What is your name?  
<Albert  
name==i1   
>secret = random.randint(1,20)  
<4  
secret==i1   
>Well, Albert, I am thinking of a number between 1 and 20.  
guess\_count==0   
>Take a guess.  
<10  
guess==i1, guess>secret # Assertions can be comma delimited.  
>Your guess is too high.  
guess\_count == 1 # Repeating the guess\_count=<integer> scheme indicates iteration.  
>Take a guess.  
<2  
guess==i1, guess<secret   
>Your guess is too low.  
guess\_count==2 # iteration  
>Take a guess.  
<4  
guess==i1, guess==secret   
guess\_count + 1 == 3   
>Good job, Albert! You guessed my number in 3 guesses!  
  
  
# User loses.  
>Hello! What is your name?  
<John  
name==i1  
>secret = random.randint(1,20)  
<3  
secret==i1  
>Well, John, I am thinking of a number between 1 and 20.  
guess\_count==0  
>Take a guess.  
<11  
guess==i1, guess>secret  
>Your guess is too high.  
guess\_count == 1  
>Take a guess.  
<1  
guess==i1, guess<secret  
>Your guess is too low.  
guess\_count==2  
>Take a guess.  
<2  
guess==i1, guess<secret  
>Your guess is too low.  
guess\_count==3  
>Take a guess.  
<10  
guess==i1, guess>secret  
>Your guess is too high.  
guess\_count==4  
>Take a guess.  
<9  
guess==i1, guess>secret  
>Your guess is too high.  
guess\_count==5  
>Take a guess.  
<8  
guess==i1, guess>secret  
>Your guess is too high.  
guess\_count >=5 # User avoids guess\_count==5, as that'd look like another iteration.  
>Nope. The number I was thinking of was 3.

From that Exemplar produces function

**def** guess4():  
 print(**'Hello! What is your name?'**)  
 name = input(**"name:"**) *# Eg, Albert* print(**'secret = random.randint(1,20)'**)  
 secret = int(input(**"secret:"**)) *# Eg, 4* print(**'Well, '** + str(name) + **', I am thinking of a number between 1 and 20.'**)  
 **for** guess\_count **in** range(0, 6, 1):  
 print(**'Take a guess.'**)  
 guess = int(input(**"guess:"**)) *# Eg, 10* **if** guess>secret:  
 print(**'Your guess is too high.'**)  
 **elif** guess<secret:  
 print(**'Your guess is too low.'**)  
 **elif** guess==secret:  
 print(**'Good job, '** + str(name) + **'! You guessed my number in '** + str(guess\_count+1) + **' guesses!'**)  
 **break  
 if** guess\_count>=5:  
 print(**'Nope. The number I was thinking of was '** + str(secret) + **'.'**)

That function is produced by the same inference mechanisms described for the prime\_number example, with the addition that questions are asked of the user at the time Exemplar is run. Those are

Immediately after lines   
 elif guess==secret:  
 print('Good job, ' + str(name) + '! You guessed my number in ' + str(guess\_count+1) + ' guesses!')  
 break  
 if guess\_count>=5:  
which transformation?   
1) print('Nope. The number I was thinking of was 3.')  
2) print('Nope. The number I was thinking of was ' + str(secret) + '.')  
2  
  
Immediately after lines   
 print('Your guess is too high.')  
 elif guess<secret:  
 print('Your guess is too low.')  
 elif guess==secret:  
which transformations? Please enter the sum of your selected choices:   
1) print('Good job, Albert! You guessed my number in 3 guesses!')  
2) print('Good job, ' + str(name) + '! You guessed my number in 3 guesses!')  
4) print('Good job, Albert! You guessed my number in ' + str(guess\_count+1) + ' guesses!')  
6  
  
Immediately after lines   
 print('Hello! What is your name?')  
 name = input("name:") # Eg, Albert  
 print('secret = random.randint(1,20)')  
 secret = input("secret:") # Eg, 4  
which transformation?   
1) print('Well, Albert, I am thinking of a number between 1 and 20.')  
2) print('Well, ' + str(name) + ', I am thinking of a number between 1 and 20.')  
2

The user’s answer to the first and last questions teach Exemplar to use variable output, not a hard-coded 3 or “Albert”. The second answer, 6, is the sum of 2 and 4, meaning that the target function should use a variable name (transformation 2) and variable number (transformation 4) when congratulating a winning player.

Those questions and the user’s answers (2, 6, 2) are stored together in a “.qa” file for subsequent retrieval by Exemplar each time a code trace file with the same name (guess4.exem) is submitted for TestGuess4.py file generation.

N.B. random.randint() is modelled as an output in the .exem and not actually called because the result of doing so would be nondeterministic absent complicating controls cost ineffective in a proof of concept.

**How Exemplar works, internally**

Each time exemplar.py is run, a code trace (.exem file) must be specified as an argument.

Then exemplar.py’s \_\_main\_\_ section launches a function named reverse\_trace() that resets the (Sqlite) database before inserting into it the code trace lines pulled from the given .exem file.

The (lightly edited) body of reverse\_trace():

reset\_db()  
example\_lines = from\_file(file)  
fill\_example\_lines(example\_lines) # Insert the .exem lines and  
fill\_conditions\_table() # user assertions into the database.store\_fors() # Similarly store info on probable loops.code, test\_file\_contents, success = **get\_function**(file) # Derive a conforming function. **return** code, test\_file\_contents

Even though data is not persisted beyond an execution of Exemplar, database savepoints and rollback is useful to start over whenever an algorithmic interpretation of the examples proves faulty (i.e., results in error or inconsistency).

After the example\_lines table is filled, fill\_conditions\_table() heuristically categorizes all the assertions as either assignment, IF condition, or FOR condition.

Then store\_fors() navigates many order-sensitive state changes to determine each FOR block’s start point, first potential stopping point, and final potential stopping point.

Processing after store\_fors() runs is orchestrated by a large function named get\_function(), which first pulls a complete set of potential FOR loop stopping points (via get\_last\_el\_id\_maybes()), then begins a database transaction and documents those potential stopping points in a temporary database table. If any of the conjectured stopping points prove untenable, the transaction is rolled back, and another set of potential stopping points is pulled.

Nested under this procedure is the equivalent process for IF controls. I.e., store\_ifs(), a function much like store\_fors(), notes each IF block’s starting point, first potential stopping point, and last possible stopping point. From this potential pool, a definite endpoint is conjectured for each IF block and another database transaction (nesting into one started by the lastest FOR loop interpretation) starts, and these endpoints are written to a temporary database table. Again, as with FOR blocks, as soon as an inconsistency is detected, the latest transaction (only) is rolled back and a new set of potential IF endpoints is trialed.

When an interpretation of FOR loops and IF blocks reaches generate\_code() (another large function), a target function is synthesized via the reasoning outlined in the preceding section (How Exemplar works, to the user).

At the end of generate\_code(), order\_IFs() reorders the target function’s IF branches until all examples pass. (get\_IF\_permutation() uses factoradics [5] to number and track the IF branch orderings.) That doesn’t involve testing the full function, just setting variable values to match a particular user example at the point of the putative IF branching point and seeing if the proposed IF branch triggered in an eval() is the right one. I.e., matches that of the user example. *provide example*

After generate\_code(), generate\_tests() creates the test suite, and the resulting unit tests are executed against the synthesized function. If all tests pass, the target function is returned. Otherwise, the latest database transaction is rolled back and the next set of control structure endpoints are trialed. This continues until no errors are detected or Exemplar fails (i.e., there are no more endpoints to try, in which case the database changes are committed for post mortem analysis).

**SUMMARY**

A metaprogram can automatically discover the code correlating to simply traced use cases, saving programming time when done well. This is true even if the resulting function only models the desired program via its correct control structures, as it should then be relatively easy for the programmer to polish the program’s, e.g., print statements.

The present work resolved PbE’s problem of ambiguity by a combination of approaches:

1. Hand written code trace assertions of use cases are required that, together, imply an algorithm.
2. Automatic generate-and-testing of code to find each control structure’s endpoint.
3. One-time multiple choice questions are put to the user where multiple generalization possibilities remain.

**References**

1. <https://github.gatech.edu/gherson3/Exemplar>
2. Solar-Lezama, A. (2003) Program Synthesis by Sketching, a Texas A&M University PhD [thesis](https://people.csail.mit.edu/asolar/papers/thesis.pdf)
3. Singh R., Gulwani S. (2015) Predicting a Correct Program in Programming by Example. In: Kroening D., Păsăreanu C. (eds) Computer Aided Verification. CAV 2015. Lecture Notes in Computer Science, vol 9206. Springer, Cham
4. <http://inventwithpython.com/invent4thed/chapter3.html> Retrieved 2/1/2019
5. Wikipedia contributors. (2019, November 8). Factorial number system. In *Wikipedia, The Free Encyclopedia*. Retrieved 05:42, December 9, 2019, from <https://en.wikipedia.org/w/index.php?title=Factorial_number_system&oldid=925250828>