Alan W. Biermann - Approaches to Automatic Programming

Tom Rochette <tom.rochette@coreteks.org>

August 30, 2025 — 861fb9d0

0.1 Context

0.2 Learned in this study

- When discussing with someone else, we slowly construct a context as the discussion goes on. When a discussion thread finishes, this context slowly fades away to give place to the construction of a new context.

0.3 Things to explore

- Is building a set of inductive/constructive examples more likely to properly induce a program synthesizer toward the appropriate program?
- What is the maximal size of a function such an could generate?
- How complex are the functions that can be generated? Can it handle recursive functions?

1 Overview

2 Notes

2.1 2 Extensions to Traditional Automatic Programming Methods

- The same patterns seem to appear again and aagin in the code (of traditional compiled languages)
- The development of higher level languages has largely been involved with discovering these patterns and designing constructs that implement them automatically
- Three kinds of patterns and languages that save the user from having to code them by hand:
  - DO-loops (or FOR-loops)
  - searching with automatic backtracking
  - the representation and handling of certain mathematical entities such as sets and relations in some higher level languages

2.2 2.2 Higher Level Languages

- Efficient compilation of code can be extremely difficult because the constructs of the language differ so greatly from the hardware capabilities of the machine
- For example, sets can be represented as linear linked lists, binary trees, bit strings, hash tables, fixed length arrays, and others, and the choice of data structure greatly affects the efficiency of the program
- Ordinarily these decisions concerning representation would be made by the programmer, who knows the nature of his data and how they should be ordered and accessed
- The higher level language compiler must either make arbitrary decisions at the risk of terrible performance or gather information about the usage of each data structure and attempt to make optimum decisions

- Low has written a compiler for a subset of SAIL that also makes use of statement execution counts obtained by running the program
- Higher level languages take the user one step farther away from the machine, enabling him to write programs more quickly, more concisely and more reliably
- Higher level languages will probably be successful to the extent that they embody the structure and processing that fit the user's problems and the user's concept of his problems
- These languages usually are considerably less computationally efficient than more traditional languages because their processors are not able to utilize completely special domain-dependent information in the way that a human coder could

2.3 3 Program Synthesis from Examples

2.4 3.1 Introduction

- Many a programmer has wished for a system that would read in a few examples of the desired program behavior and automatically create that program
- The synthesis task itself from weak input information is incredibly difficult
- The program synthesizer could attempt to do its job by enumerating the set of all possible programs in the language in order of increasing length, testing each one to see if it is capable of the desired behavior
- When it finds such a program, it prints it out as its answer and halts
- We know that the correct answer will exist somewhere in the enumeration so that it will be found eventually
- But this strategy has a severe pitfall because it is not possible to tell for an arbitrary program whether or not it can produce the desired behavior
- Since the halting problem for an arbitrary program on given data is undecidable, one cannot tell what the program might print out, since one cannot tell whether it will even halt
- This problem cannot, in general, be avoided, so that we have a theorem:
  - The programs for the partial recursive functions cannot be generated from samples of input-output behavior
- The only way this strategy can be made to work is to enumerate a subset of the partial recursive functions for which the halting problem is solvable or to allow for a partial solution to the halting problem by limiting the number of steps a program may complete before it halts
- Suppose the solution by enumeration method can be made to work. It is still possible that the system might produce a wrong answer because some program that precedes the correct answer in the enumeration might be able to complete the given examples
- The severest difficulty with the solution by enumeration is the extreme cost of the enumeration
- The target program in (the current example) would be beyond the billionth program in most enumerations, meaning that a great amount of time might pass before even this trivial program could be found
- Therefore, the task of program synthesis from examples is essentially intractable unless these problems can be avoided
- Two methods for making the approach feasible:
  - Limit the class of synthesizable programs
  - Include intermediate information about how each output is obtained from its corresponding input

2.5 3.2 The Method

- A reasonable technique for generating programs from examples of their behavior executes the following two steps:
  - For each example input X_i and its associated output Y_i, determine the sequence S_i of operations required to convert X_i to Y_i
  - Find a program that executes each required sequence of operations S_i when given its associated input X_i
- The discovery of acceptable sequences S_i can involve an astronomical amount of enumeration

- There may be many sequences S_i that convert X_i to Y_i and a method must be found for discovering which sequence to use for each i.

2.6 3.6 Discussion

- Program construction processes have been speeded up sufficiently through the use of enumeration pruning, limiting assumptions on the class of programs being synthesized, and user-specified example calculations so that programs of practical size can be created automatically
- The approach (described) has the following advantages:
  - The user has no need to learn traditional language syntax
  - The user has direct visual contact with his data structures and can manipulate them in an extremely natural manner with his hands
- Major disadvantages of the approach are:
  - The display terminals are not large enough to display easily large or multidimensional data structures
  - The correctness of automatically generated programs is not always easy to determine
- The "trace" of the computation can be efficiently constructed for large classes of functions from the structures of the input and output lists
- Furthermore, the programs can be built from the traces almost algorithmically with little or no searching

2.7 4 Synthesis from Formal Input-Output Specifications

2.8 4.1 Introduction

- Rather than giving examples of the desired program behavior, it may be preferable to specify precisely the required input-output characteristics and have the program automatically generate from these

2.9 4.3 Problem Reduction Methods

- P{A}Q means that if assertions P are true and program A is executed to halt, then Q will be true
- The program synthesis problem then can be stated: Given input specification I and output specification G, find A such that I{A}G
- A typical reduction step would be to divide A into two segments A1 and A2 and attempt to construct A1 and A2 separately
- A set of intermediate specification Q are determined and the two problems I{A1}Q and Q{A2}G are attacked separately
- The Buchanan and Luckham approach assumes that the system has a large amount of programming knowledge in the form of inference rules. The system also needs domain specific information, called frame information

2.10 5 Translation of Natural Language Commands

2.11 5.2 Syntatic Analysis

- Systemic grammars hierarchically decompose utterances into three basic classes:
  - clauses
  - groups
  - words
- Each sentence is broken down into one or more clauses, clauses are primarily made up of groups, and groups are primarily composed of words
- There are four types of groups:
  - noun
  - verb
  - preposition
  - adjective

2.12 6 Heuristic Knowledge-Based Algorithm Synthesis

2.13 6.1 Introduction

- A program synthesis is said to be sound if the program produced is guaranteed to meet the specifications input to the system
- A system is said to be complete if it is capable of producing every possible program over the domain of interest (usually the partial recursive functions)
- It is desired that the system should have
  - programming knowledge such as how to declare data structures, build loops and branches, and so forth
  - problem domain knowledge such as what are the significant variables and how are they related
  - debugging knowledge such as how to discover and remove the cause of a discrepancy between program performance and specification
  - knowledge of the user such as what information to expect from him, what information to send him, and how to converse with him in his own language
- A heursitic program will be defined as a program whose input-output characteristics are not easily specified (except perhaps by paraphrasing the program itself)
- While the program synthesizer using heuristics might produce false starts, try various ideas, modify partial solutions, and erase and begin again, it is hoped that it can slowly converge to a reasonable solution, particularly if it can work continuously in an interaction with a human being

2.14 6.2 The Major Phases of Processing

- Balzer has described the automatic programming process as being divided into four major phases:
  - problem acquisition when the system interacts with the user to build a model of the problem domain and the problem to be solved
  - process transformation when the problem-relevant portions of the model are sifted out to obtain an efficient representation for problem solution
  - model verification when testing and debugging are done to check whether the abstracted model is correct
  - automatic coding when the program is actually generated

2.15 6.3 Actors, Beings, Frames, and Others

- Two kinds of modularity:
  - Modules of knowledge
  - Modules of programming
- Modules of knowledge should have a body of information that can be referred to (arrays have dimensions, they may have a type, a declaration may be required in the program, arrays are sometimes initialized at the beginning of a program, they are scanned using nested FOR loops, etc.)
- Modules of knowledge must necessarily have default values (things that are automatically assumed to be true)
- Programming modules composed of relatively independent entities, which activate themselves and which send and receive information without prompting
- The control structure of traditional large program with its well-defined hierarchy of subroutines is being abandoned and being replaced by a kind of democracy of routines
- Each routine has knowledge about what it can do, what it needs to know, when it can function, how much work it may have to do, what other routines may be able to help it, and other things *(this sounds a lot like OOP, but it is not)*
- Lenat speaks of a "community of experts," and if a problem is made available to them, each one comes forth to contribute knowledge and help if he is able
- Instead of an individual routine receiving x and sending back a value f(x), the whole group of routines might receive the request: Does anyone know anything about "adding"?
- Several routines might respond

- The request may have to be followed with more information until one routine sees its own applicability and takes control
- The attractions of this type of programming are many. It enables the programmer to delay decisions about the control structure until knowledge modules are being constructed and to base control decisions on the contents of these individual modules *(I see this as being able to choose between various sorting algorithms, it's interesting, but you generally have a preference over which one you want to use)*
- It makes it possible to design individual modules without as much concern for their effect on the rest of the system *(that is already the case when you write a couple of functions that are never called... I believe the idea here is more about planning for the future, than writing code for current needs)*
- The problem with such code, of course, is that upon being given a task to do, the group of routines may be contented to sit there and send messages back and forth without ever making any progress

2.16 6.4 On the Development of Knowledge about Knowledge

- The study of knowledge:
  - how it is represented
  - how it is acquired
  - manipulated
  - accessed
  - how it is used to create programs
- The most important problem to be addressed is the representation problem: How is knowledge to be represented?
  - Facts could be stored in the machine in terms of tables, property lists, semantic nets, formal logic axioms, executable programs, and many other forms
- One might ask how much knowledge is required to do a particular task
  - The required amount of knowledge could be measured in terms of numbers of facts or perhaps number of beings
- Another recent interest is the study of approximate knowledge and its use and modification
- Sussman has his system propose a program to solve a problem, even though the program may not be at all correct. Then his system modifies the first approximation until it converges on a solution
- How is knowledge about knowledge to be obtained? The answer from the artificial intelligence community seems to be unanimous: One should study examples
- One should look sequentially at how several similar programs might be produced, how a class of related programs might be produced, and eventually how the synthesis capability might be extended to other classes of programs (produce solution specific programs, then indicate what may be generalized)

2.17 6.5 Summary

- The goals of the designers of these systems are ambitious:
  - to build in a natural language understanding and generation ability
  - to incorporate a model-building function and non-trivial problem solving abilities
  - to include learning and inductive abilities

2.18 7 Comments

- An examination of the literature seems to indicate that there are exactly three basic processes that a system can use in obtaining the desired program:
  - The system can be directly given the program or information from which the program can be directly built. Thus the program might be typed in by the user in a language that can be directly converted to the target program, or the target program may already exist in a library
  - The system may have to enumerate from the set of all possible programs, from the set of all possible proofs, or from some other space until an acceptable answer is found
  - The system may be able to build the desired program by modifying and combining known programs

- Research in automatic programming involves a study of the languages of the human mind, the languages of machines, and the process of translating between the two
- This author views the main task in automatic programming to be the discovery of the nature of these languages and the clever implementation of the three given processes to do the translation

3 See also

4 References

- doi:10.1016/S0065-2458(08)60519-7