What Can Automatic Programming Learn from Theoretical Computer Science?

Colin G. Johnson
Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent, CT2 7NF, England
C.G.Johnson@ukc.ac.uk

Abstract

This paper considers two (seemingly) radically different perspectives on the construction of software. On one hand, search-based heuristics such as genetic programming. On the other hand, the theories of programming which underpin mathematical program analysis and formal methods. The main part of the paper surveys possible links between these perspectives. In particular the contrast between inductive and deductive approaches to software construction are studied, and various suggestions are made as to how randomized search heuristics can be combined with formal approaches to software construction without compromising the rigorous provability of the results. The aim of the ideas proposed is to improve the efficiency, effectiveness and safety of search-based automatic programming.

1 Introduction

In recent years a number of systems have been developed which apply computational intelligence techniques to the automated creation of computer programs. Many of these have been based on biologically-inspired methods such as genetic algorithms: genetic programming [3, 11, 21] (and the many variants thereon)is the most widely used of these. By looking at further details of evolutionary biology various extensions of these ideas have been created, such as grammatical evolution [30], which exploits the idea of gene expression. Other system have used tabu search as a way of exploring program-space [6], or have developed search techniques which are most closely tied into program structure [29]. These techniques have been shown to be successful on a wide variety of problems, though the success of the various techniques on different kinds of problems is variable. For the purposes of this paper we shall refer to such search-based, heuristic methods of creating programs as *automated programming* systems.

Alongside this development there continues to be a substantial development in many areas of theoretical computer science concerned with understanding programs, the way in which programs can be constructed and how properties of programs can be verified.

At first these two areas of study appear to be completely at odds with each other. On one hand is the sloppy, heuristic world of GP and its fellows, seen typically as unrigorous but potentially effective at solving the sort of ill-defined, fuzzy problems which are found in “the real world”. On the other hand the theoretically rigorous work is seen as very precise and exact; however this work is often seen as not being flexible enough to be applied to realistic problems.

However seen from another perspective the two approaches seem to have much in common. Human programmers often have a view of code as a brittle object where the slightest erroneous change can lead to catastrophic error. By contrast to this both heuristic-driven automated programming and many of the theoretical approaches to programming view code-stuff as a malleable material, able to be transformed and restructured by a wide variety of transformations without breaking it.

This paper surveys some of the potential for cross-fertilization between these areas. The aim is to spark interest in this area through a broad survey of ideas and suggest topics which might provide a bridge between the theory of programming and comptuational intelligence techniques. The basic question addressed is this: how can an understanding of the theoretical perspectives on programming make automatic programming more effective, efficient and safe?

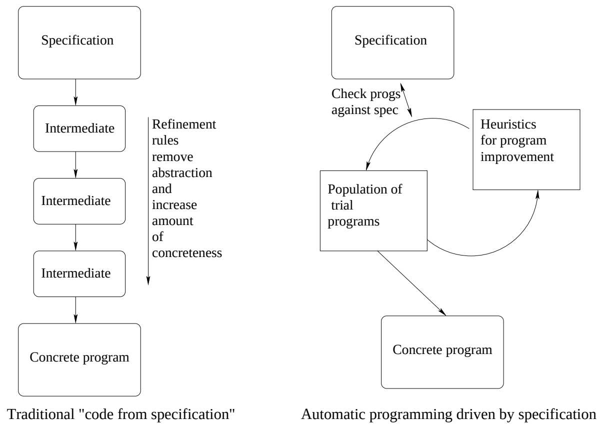

Figure 1: Contrasting inductive and deductive approaches to the relationship between programs and specifications.

2 Program Induction vs. Program Deduction

One way to create a unified framework in which to view these diverse approaches is to consider the relationship between specifications and programs. One stance towards the creation of code is that programming is the process of transforming *specifications* of desired behaviour into programs which carry out that behaviour. By *specification* we mean any means by which we can express that behaviour. This could be a formal specification written in a specification language, but could also be some informal idea of what the program is intended to achieve (an interesting discussion of what "specification" might mean is given in [32]). For a program to implement a particular specification everything asked for in the specification must be implemented in the program, and any constraints in the specification must be respected by the program.

From a traditional "formal methods" perspective this process has been viewed as a *deductive* problem. A set of operations is defined which describe how to transform a specification statement into a concrete statement which realizes that specification step. This can be enriched by the inclusion of operations which use operations which transform some part of the specification into some intermediate state, less abstract than the full specification but not fully executable. An example of such a program deduction process is *refinement* [26].

In contrast to this the automatic programming systems described above take an *inductive* stance towards program creation. A program induction system consists of two components. The first is a way of measuring the closeness of programs to specifications, and the second is a way of using that information to suggest ways in which programs might be brought closer to the specification. Another way to state this is that the algorithm is *searching* problem space for a program which matches the specification [16, 17, 29].

However program induction has its own problems. Typically the measure used to decide whether programs are close to their specification consists simply of running the programs on test data and measuring the distance between the solutions produced by the program and the solutions acceptable to the specification using some metric in solution-space. Whilst this has been empirically shown to provide a powerful way of directing program towards appropriate parts of the program-space, it gives no formal confidence that the programs induced will apply anywhere other than on the test data sets used.

This dual perspective allows us to place inductive and deductive techniques side-by-side and study them for commonalities and ways in which we might combine the two perspectives. A summary of the two views is given in figure 1.

3 How Might Programming Language Theory Inform Automatic Programming?

This section looks at a number of areas of programming language theory and suggests ways in which these might interact with automatic programming.

3.1 Static Analysis and Model Checking

Static analysis [27] is an overarching term for a range of techniques which extract information out of programs without explicitly running them on particular data sets. A typical technique is that of *abstract interpretation* [8, 9], in which the data processed by the program and the various operators within the programming language which change those data are "abstracted" into sets of properties of interest. The analysis of the program consists of tracking whether these properties are guaranteed to hold at any particular point in the program. That is, the analysis makes a conservative approximation to whether a particular property holds or not. A large number of different kinds of information can be extracted from programs by this kind of analysis, for example:

- constraint information about relationships between variables [10]
- information about the extreme values which a variable could possibly take at each point in the execution of a program [8]
- usage information about whether facts vital to the solution of a problem have been used
- complexity information, e.g. the number of potential paths through a piece of code [23]
- performance information [28, 35]

Consider for example an integer variable within a program. In a normal run of a program a particular initial value is assigned to that variable and various operators act on the variable to change that value. In a static analysis of a program a *property* of the variable is followed through the program. To illustrate this we shall consider two examples of the kind of information which can be tracked through a program using a static analysis.

For each example we give four pieces of information: the condition which is being tracked through the program the values which that condition can take, the initial value which it takes when a variable is defined without giving it a value, and some examples of update rules which show how actions in the program affect the abstract variable.

The first example is whether a given integer variable is guaranteed to be ≥ 0.

Condition: An integer variable x is positive or zero.

Values: True (T) or don't know (D).

Initial value: D.

Update rules: Some examples:

- If a positive constant is assigned to x then value becomes T.
- If x becomes equal to something which is known to be positive-or-zero then value becomes T.
- If an integer is added to x then value becomes D.
- If value is currently T and something which is known to be positive-or-zero is added to x then value remains at T.
- If x becomes equal to the absolute value of any expression then value becomes T
- If any value is subtracted from x then value becomes D.
- ...and so on ...

Here is another example; in this example we track the upper and lower bounds on the integer value.

Condition: The upper and lower bounds an integer variable x can take.

Values: A pair of integers x_{min} and x_{max}, where we can be confident that $x \in [x_{min}, x_{max}]$.

Initial value: $[-\infty, +\infty]$.

Update rules: Some examples:

- If a constant value y is assigned to x, then $x_{min} := y$ and $x_{max} := y$.
- If a constant y is added to x, then $x_{min} := x_{min}+y$ and $x_{max} := x_{max}+y$.
- If the absolute value operator is applied to x, then if $x_{min} < 0$ we can make $x_{min} := 0$.
- If a variable which is known to be positive-or-zero is added to x then $x_{max} := \infty$.
- ...and so on ...

In developing such an analysis program we begin with simple, clearly true statements, and gradually "refine" them to produce more accurate statements. E.g. in the first example above we had the following statement:

If any value is subtracted from x then [the positive-or-zero] value becomes D.

this is a conservative approximation: consider the following code.

int x, y
input a value in the range $[0, 20]$ into x
input a value in the range $[0, 5]$ into y
$x := x + 10$
$x := x - y$

Applying the rule above makes a true statement, as long as we remember that "don't know" doesn't mean "can't know". However this update rule can be refined as follows:

If any value is subtracted from x then value becomes D, unless $x_{min} - y_{max} > 0$, in which case value becomes T.

Now note that by the end of line 4 we know that $x \in [10, 30]$ and $y \in [0, 5]$, so $x_{min} - y_{max} > 0$, therefore we can assign T as the positive-or-zero value for x.

By using such techniques we can discover whether particular properties hold, either throughout a program or at the end of the program. This is typically seen as a way of deducing information about a program which is in the debugging or testing stage. However in the context of automatic programming we can see these techniques as ways of measuring the fitness of programs [18, 19]. If we can specify certain desired properties of programs in terms of such properties then a static analysis can potentially give a guarantee that a program must satisfy that property regardless of input. This would seem to be particularly important in imposing safety constraints. In many situations it is important that a variable (whether a variable in a program or a derived quantity) is bounded within a certain range. For example a robot may be constrained so that no movement takes it outside its working area, or the temperature of some process must not exceed some critical value. The fitness function in these cases is not based around testing but based around checks to confirm whether these constraints are guaranteed to hold.

Importantly there is the potential to use multi-criterion optimization to combine essential fitness constraints with desirable, data-driven features. Many problems can be partially specified by a set of formal statements about the variables and their relationships, whilst other aspects of the problem can only be expressed as data. Combining satisfaction of a number of statically combined constraints together with optimization of some data-defined features into a multicriterion fitness function could be a way of satisfying both of these simultaneously.

Many of the above ideas could also potentially be implemented using the ideas of *model-based systems* and *qualitative reasoning* [22, 34]. These techniques are also concerned with tracking properties of variables through a program.

A related topic is *model checking* [7]. One important check which model checking provides is a check as to whether a temporal logic formula holds over a given system, specified using e.g. an finite state machine or Petri net. Again this has the potential to be used as a fitness driver for the creation of such systems—the idea of evolving finite state machines has recently undergone a revival [5, 36] after being one of the earliest applications of evolution-like algorithms [14]. Interestingly the output of a model checking procedure which fails is a specific counterexample to the statement; some automated analysis of this could be carried out to determine which direction in the search space might prove profitable. Also it may be possible to make a graded sequence of logical formulas which guide evolution towards the solution; the final desired set of constraints could be relaxed into a set of less-strict constraints, and the strictness annealed with time until the final constraints are satisfied.

3.2 Specifications and Refinement

A substantial area of theory which lies at the heart of *formal methods* is the idea of *refining* a specification. A refinement [13, 26] of a specification is the replacement of various abstract statements with more concrete statements. The set of allowable replacements is formalized in a set of rules called a *refinement calculus*. As these rules are applied repeatedly, the abstract parts of the specification are eventually replaced by concrete, executable statements which eventually (provided the refinement is successful) provide an executable program.

There are several points in this process where a search-based automatic programming approach might be used. Firstly it may be possible to use a search algorithm to decide which sequence of refinements to apply. Less obviously this may provide a way of combining data-driven and specification-driven aspects of a program. One problem with automated refinement of programs from a specification is that all of the information about the program must be contained in the specification. In many cases problems have a dual nature: we are able to provide a specification for some parts of the desired functionality, whereas the remaining functionality is provided by data. Indeed, as pointed out by Partridge [31, 33], some problems (or parts of problems) are *defined* by sets of data.

One of the difficulties in this sort of problem is in assigning fitness values to partially-concrete programs. We cannot execute these in a conventional fashion as parts of the specification remain abstract. One promising approach to this is the *executable specifications* of Barnett *et al.* [4].

3.3 Program Transformation and Mutation Testing

One of the important features of automatic programming is the need to make meaningful transformations of programs. For example at the core of GP are recombination and (typically less importantly) mutation operators. In the theory of programming there is also much work about how programs can be transformed [25]. These two kinds of transformation have somewhat different aims. In the theory of programming the transformations are typically applied with the aim of preserving the semantics of the program;

the aim of the transformation is to make the program more specialized and therefore faster on a particular problem, or to overcome some hardware constraints by rearranging the structure of the program so that it compiles in such a way that it respects the underlying hardware structure (an example is given in [12]). In automatic programming the aim is to transform a program so that it makes a move through the space of program semantics; however that move is not an arbitrary move, we want to ensure that the move has certain properties.

For example the desired outcome from a mutation operation is a "small move" to a program which is close in meaning to the original program, whilst a recombination operation is designed to produce a child program which includes some aspects of the semantics of the two parent programs. Can we make use of the understanding which the theory of program transformations gives us about how changes to code affect the meaning of programs to construct appropriate move operators?

A related area of interest is *partial evaluation* of programs [20, 24]. The aim of partial evaluation is to transform a more general program into a more specialized one, typically with the aim of making a program more efficient. A toy example would be taking a program which takes two variables as input, fixing the value of one of the input variables, substituting an appropriate value throughout the program, then recompiling the program with the fixed value. This would mean that the program would run much more efficiently on the restricted input space; many of the calculations and branching decisions which would have to be taken (relatively slowly) at runtime will have already been taken at compile-time and so will execute relatively faster. This eliminates a traditional tradeoff between writing generic software which carries a lot of runtime baggage with it and the human cost of writing specialized code for each area of application. This has been applied e.g. to the specialization of Fast Fourier Transforms for a particular application area [15] and to the specialization of a ray tracing program with respect to various inputs, e.g. the scene, the lighting conditions, and the point of view [2].

Again there are many potential connections between automatic programming and partial evaluation. Can we use search algorithms to automatically specialize a given program so that it is appropriate for a class of problems implicitly defined by a set of training data? Can a set of suitably generic algorithms be combined together and specialized appropriately in a GP-like system?

4 Randomness and Reliability

One criticism of heuristic methods is their dependency on randomized search as part of their discovery engine. However the use of randomization doesn't necessarily rule out the production of end-products which rigorously satisfy a set of requirements.

The above approaches ensure that the end results are still reliable despite the use of randomness in a number of ways. Some of them begin from an abstract but correct version of the problem, and ensure that the search process never makes a move outside this correct region of search space. Others use conservative approximations to check whether the programs generated by the system fall inside the requirements. An alternative approach would be to apply search algorithms simultaneously to the solutions themselves and to proofs of required properties about those solutions.

5 Conclusions

We have surveyed several different approaches to the formal analysis of programs from the perspective of automatic programming. A number of ways in which these two seemingly diverse subjects can be combined are suggested. In particular many of these techniques provide new ways of defining fitness functions, particularly for safety-critical systems, and they can offer insights into how moves in a search algorithm change the meaning of programs.

Acknowledgements

Thanks to Andy King, Simon Thompson, John Clark, James Foster and Howard Bowman for conversations on these topics.

References

[1] *Proceedings of the 2000 Congress on Evolutionary Computation.* IEEE Press, 2000.

[2] P. Andersen. Partial evaluation applied to ray tracing. Technical Report D-178, University of Copenhagen Department of Computer Science, 1993.

[3] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An*

Introduction. Morgan Kaufmann, 1998.

[4] M. Barnett, C. Campbell, W. Schulte, and M. Veanes. Specification, simulation and testing of COM components using abstract state machines. In R. Moreno-Díaz and A. Quesada-Arencibia, editors, *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 266–270, 2001.

[5] K. A. Benson. Evolving finite state machines with embedded genetic programming for automatic target detection within SAR imagery. In *Proceedings of the 2000 Congress on Evolutionary Computation* [1], pages 1543–1549.

[6] J. A. Clark and J. L. Jacob. Protocols are programs too: the meta-heuristic search for security protocols. *Information and Software Technology*, 43(14):891–904, 2001.

[7] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[8] P. Cousot. Abstract interpretation: Achievements and perspectives. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 224 and electronic proceedings `http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`, L'Aquila, Italy, July 31 – August 6 2000. Scuola Superiore G. Reiss Romoli.

[9] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[11] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. J. Greffenstette, editor, *Proceedings of the first international conference on genetic algorithms and their applications*, pages 183–187. Erlbaum, 1985.

[12] S. Debray. Resource-bounded partial evaluation. In *Proceedings of the 1997 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pages 179–192, 1997.

[13] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.

[14] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. Academic Press, 1966.

[15] R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Dolevzal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman and Hall, 1995.

[16] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[17] M. Harman and B. F. Jones. Software engineering using metaheuristic innovative algorithms: workshop report. *Information and Software Technology*, 43(14):905–907, 2001.

[18] L. Huelsbergen. Abstract program evaluation and its application to sorter evolution. In *Proceedings of the 2000 Congress on Evolutionary Computation* [1], pages 1407–1414.

[19] C. G. Johnson. Deriving genetic programming fitness properties by static analysis. In J. Foster, E. Lutton, C. Ryan, and A. Tettamanzi, editors, *Proceedings of the 2002 European Conference on Genetic Programming*. Springer, 2002.

[20] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[21] J. R. Koza. *Genetic Programming : On the Programming of Computers by means of Natural Selection*. Series in Complex Adaptive Systems. MIT Press, 1992.

[22] B. Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, 1994.

[23] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989.

[24] T. Mogensen and P. Thiemann, editors. *Partial Evaluation—Practice and Theory.* Springer, 1998.

[25] T. Æ. Mogensen. Evolution of partial evaluators: Removing inherited limits. In *Proceedings of Partial Evaluation'96*, Lecture Notes in Computer Science vol. 1110. Springer, 1996.

[26] C. Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[27] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.

[28] K. D. Nilsen and B. Rygg. Worst-case execution time analysis on modern processors. In *ACM PLDI Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[29] R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–83, 1995.

[30] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, August 2001.

[31] D. Partridge. Non-programmed computation. *Communications of the ACM*, 43(11):293–302, 2000.

[32] D. Partridge and A. Galton. The specification of 'specification'. *Minds and Machines*, 5(2):243–255, 1995.

[33] D. Partridge and W. Yates. Data-defined problems and multiversion neural-net systems. *Journal of Intelligent Systems*, 7(1–2):19–32, 1997.

[34] C. J. Puccia and R. Levins. *Qualitative Modelling of Complex Systems.* Harvard University Press, Cambridge, MA, 1985.

[35] P. Puchner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1:159–176, 1989.

[36] W. M. Spears and D. F. Gordon. Evolution of strategies for resource protection problems. In A. Ghosh and S. Tsutsui, editors, *Advances in Evolutionary Computation: Recent Trends.* Springer, 2002.