LOGIC ORIENTED PROGRAM SYNTHESIS

goals and realization

This work has been funded by ESPRIT

Christoph Kreitz
Gerd Neugebauer

Institut für Informatik
Forschungsgruppe Intellektik
Technische Universität München
Arcisstr. 21
D-8000 München 2

Bertram Fronhöfer

LIFIA - INPG
46, Av. Felix Viallet
F-38031 Grenoble Cedex

ABSTRACT:

Automated Program Synthesis from logical specifications nowadays has to face attacks coming from two directions. "Real" Programmers often argue that it is a nice academic toy, good to generate a handful of small examples, but of no use at all in the hard real world of software technology. On the other hand, due to the advent of Logic Programming the distinction between specification language and program language got blurred and some people believe that Program Synthesis has become an obsolete field of research. In view of the existing papers and systems both opinions appear to be quite natural. Therefore it has to be clarified where the real great challenges to Program Synthesis are today.

Our paper intends to open the discussion on the topic again by expounding a view of the field which arose from experiences with implementing a Logic Oriented Program Synthesizer (LOPS). After stating what the tasks of a Program Synthesis System should be we will give a methodological guideline for the practical realization of such a system. By following our suggestions, we believe, both attacks against the field can be countered successfully. Firstly, Program Synthesis will indeed have useful applications in industry and secondly, as we will show, Logic Programming languages are not at all a solution to the problems of the field.

1. INTRODUCTION

Already in the early days of Computer Science programming methodologists have devoted themselves to develop strategies of structured programming that will ensure the correctness of the resulting program. Beginning with a mathematical description of the relation between input and output the technique of stepwise refinement shall eventually lead to a program fulfilling this specification. Since then, attempts have been made to formalize and eventually automatize the programming process (see e.g. [BD 77], [MW 77], [BIB 80], [HOG 81], [DER 85]). A programmer should be allowed to specify his problem in a purely logical way while being freed from the burden of writing down explicit computer code by himself. Having to care about the peculiarities of a specific programming language is always an annoying and often tiring task and many mistakes result from the lack of concentration caused by that. The ultimate goal of traditional Program Synthesis was the automated construction of correct programs just from their specifications.

Due to the difficulty of the subject Program Synthesizers haven't been extremely successful yet as far as the possibility of industrial application is concerned. Fully automated generation of programs for hard software problems currently appears too big a task. It therefore has been argued that Program Synthesis is but a nice academic toy, good to synthesize some trivial example algorithms like sorting lists or computing their maximum. In the world of real programmers, however, it would be of no use at all.

On the other hand, lots of efforts have been put into the development of Logic Programming Languages. Programs now can be written in a logic-like style and the distinction between program and specification seems to become obsolete as the performance of theorem provers increases continuously. The traditional task of Program Synthesis would disappear then and make the whole field of research meaningless.

Threatened from two sides Program Synthesis needs to restate its goals for today. Where are the real great challenges in program development - is there a task that goes beyond what Logic Programming Languages can offer? Does Program Synthesis have anything to offer to the real programmer's world? We believe that both questions can be answered positively. As shown in the next chapter the question how to develop good programs contains more than just the problem of correctness. Also, Logic Programming Languages are still programming languages with peculiarities and thus can not provide a full answer to it. However, they have certain advantages and we should make use of them instead of defending ourselves against them. We believe that the future of Program Synthesis lies in logic oriented systems, i.e. systems that will strongly support a user in writing "good" logic programs.

In the following chapter we will try to give a new meaning to the original goals of the field by presenting our view of the tasks a (logic oriented) Program Synthesis System should be able to perform. After reviewing some of the traditional approaches to the field we then give a methodological guideline how to realize such a system in a structured way. As an example in the last chapter we briefly describe the (new) design of a Logic Oriented Program Synthesizer called LOPS which is based on these considerations. In order to open the discussion to a wider range of people we chose to avoid formality in this paper.

2. THE GOALS OF LOGIC ORIENTED PROGRAM SYNTHESIS

Principally, the goal of Program Synthesis is to support a programmer in developing good programs from (mathematical) specifications while freeing him from having to care about peculiarities of a specific programming language.

In view of recent developments todays interpretation of this goal has to be different from the one given 10 years ago. We therefore will state here what supporting a programmer should mean and where we expect the real challenges to the field today. The points we present are not necessarily new but important in our eyes.

Correctness of the resulting algorithm

Though assumed to be of lesser interest for current resarch activities this is still a crucial point. A tool supporting the development of programs **must** guarantee that the result correctly meets the specification. It is better to get no result at all than a false one.

Since a mathematical specification of a problem may already be viewed as some kind of Logic Program one might assume that correctness is achieved by simply using Logic Programming languages. However, in languages like PROLOG the semantics of a program strongly depends on the order of statements. It may greatly differ from the semantics of the logic formula the user unfamiliar with the peculiarites of PROLOG believed to write down. Thus using "Logic Programming" languages is not the same as Programming in Logic and even less the answer to the correctness problem.

Efficiency of the resulting algorithm

Program Synthesis shall encourage the programmer to use logic to express his problems and free him from other considerations. However, programming in an elegant and comprehensible style is often paid by a high program running time and we cannot ignore this problem if there shall ever be an industrial application for Program Synthesis. One of the great challenges, therefore, will be to find systematic ways ("strategies") how to change a logic program - the specification - into a more efficient equivalent one.

Thus Program Synthesis today will strongly resemble the technique of Program Transformation. Nevertheless we will avoid this term because it has mainly been applied to transformations of programs written in a real programming language instead of pure logic.

Completeness - no restriction to the class of problems that can be handled

Due to theoretical limits a fully automated synthesis of all problems that can be specified will not be possible. We have to expect a tradeoff between the class of algorithms that can be synthesized and the degree of automation involved. In our opinion preference should in any case be given to a synthesizer that can handle all kinds of problems regardless whether they can be automatically synthesized or not. Supporting program development, as we will see, goes beyond just solving a problem completely mechanically. Thus there is no reason why Program Synthesizers should have to totally reject a problem.

Cooperation with the programmer instead of replacing him

Since we do not believe that human intelligence can ever be fully simulated by a computer system one cannot intend to use a Program Synthesizers as a replacement for a professional programmer in the field. For this reason and as a consequence of our above demand for completeness we plead for systems cooperating with a programmer. We see no reason why the great potential of human intuition should not be used for the synthesis process.

Already there are some purely interactive systems that can claim extraordinary results in developing programs (e.g. [CIP 85]). Basing Program Synthesizers on an interactive mechanism appears to be very promising for application in the real programmer's world. Of course, the goal of Program Synthesis is to provide more support than just an formula manipulation tool. Still, an important research task is to develop heuristics guiding the synthesis strategies which will take as much as possible of the burden away from the user.

Teaching logic as a programming language

As cooperation with the programmer can improve the system's capabilities so working with the Program Synthesizer can improve the programmer's skills in programming in logic. Therefore, the aspect of teaching should become an important side-effect (or even a new task) of Program Synthesizers. Simple problems will be solved by the system and a beginner may acquire an understanding of logic programming methodology by studying the deductions performed during the synthesis. For specifications whose syntheses require interaction with the user he will have to present his own ideas and discover which methods will lead to a satisfying result and which ones do not. Since the system does not allow false solutions he will be forced to to acquire more expertise by himself in order to succeed.

Program Synthesis under the above considerations would be a major step towards using (true) logic as a programming language which would be the natural consequence of the previous evolution of programming languages from machine language to modern high level languages.

3. REALIZATION OF PROGRAM SYNTHESIS SYSTEMS

Some researchers neglect the fact that strategies and heuristics which guide the basic mechanisms of a Program Synthesis System cannot be developed purely theoretically. Practical experimentation with machines that do not allow handwaving arguments is required for getting insights into their true strengths and weaknesses. Also, only systems that run can convincingly demonstrate that in fact these concepts yield more than a toy.

Systems built and used for this purpose, after some time tend to get quite bulky and offer great problems of maintenance. We therefore propose a methodological guideline to a structured implementation of a Program Synthesizer that arose from investigating some of the existing systems and from our experiences with our own system. Since we believe that Program Synthesis Systems, particularly their heuristics, have to grow with experience our goal is to design systems to be well structured, easy to maintain, easy to expand, and of course designed to become user friendly.

We will first give a brief review on the approaches to program synthesis so far.

3.1 Traditional approaches to Program Synthesis

A synthesis problem is specified by stating the properties which input and output variables must fulfill and the relationship between them. A general specification format would be something like "given ***iv*** compute ***ov*** such that ***R(iv,ov)*** holds", where ***R(iv,ov)*** is some logical formula. Traditionally, the task of Program Synthesis Systems was interpreted as to automatically derive a program which exactly does what the specification demands.

Two principal approaches have been pursued:

The specification is rephrased as a theorem of the form *"*ALL ***iv*** EXIST ***ov R(iv,ov)"*** and the Synthesis task is precised as proving this theorem constructively, such that a functional program can be extracted (see e.g. [MW 80], [FRA 84], [BC 85]).

Another rephrasing would be the definition of a new predicate: ***P(iv,ov)*** :<-> ***R(iv,ov)***. This definition is treated as a theory and Program Synthesis means deriving theorems which are computationally convenient and therefore can be viewed as a logic program (see e.g. [BD 77], [MW 79], [HOG 81]).

These two different ideologies have various consequences:

Synthesis methods (construction/deduction versus transformation)

Pursuing the first approach the focus lies on the development of theorem provers suitable for the constructive proof of existential formulae. On the other hand, the second approach concentrates on the development of good transformational ideas and strategies to control the synthesis process while the role of theorem proving reduces to an important, but secondary one.

Problem specification (theorem versus equivalence knowledge)

In the second approach the specification defines an input-output relation and it is straightforward to say a program is equivalent to this specification if it computes the same relation. But what does it mean for a program to meet the specification (a)? Does it have to compute only one set of output values ***ov*** for a given set of input-values ***iv*** (functional program), or does it have to compute them all?

Another aspect is concerned with the predicate symbols occuring in a specification. If for some ***iv*** there is no ***ov*** such that ***R(iv,ov)*** holds, then the specification (a) is not a valid theorem and therefore cannot be synthesized. The second approach does allow to specify "partial" programs. They would simply return false in the above case and yield correct results in the other ones. However, for functional, total programs failure to prove their specification in (a) is useful for detecting errors in the specification itself.

Final step (program extraction versus control generation)

Having eventually accomplished a constructive proof in (a) a program must be extracted. This step is often already intertwined with the proving process. In the other approach, when computationally good theorems are deduced still some kind of control has to be generated.

3.2. A methodological guideline

The first one of the above approaches leads to special purpose synthesizers which may perform their task more efficiently than a general system would do. However, it neither can be applied to all kinds of programming problems nor seems it to be suited for the efficiency task. Also its mechanism for program development differs severely from the method human would use and thus it will be less appropriate as a support tool. Since the second approach one does not have any of these problems we think it a legitimate consequence to plead for a construction of Program Synthesizer along the (b)-line. Nevertheless, our suggestions below can be used for building synthesizers based on the other method as well.

Start with a fully interactive tool, i.e. a system containing a small set of parameterized rules transforming logic formulae that may be applied interactively. These transformation rules will be considered as the basic (sub-) strategies of the system. To ensure correctness these transformations must be equivalence preserving.

This simple tool already would be a great support for a programmer because it allows to experiment with different kind of transformations without having to calculate the exact result by hand and also guarantees the correctness of any result that will be obtained with it.

On top of the above tool a simple but powerful strategy should control the synthesis process by determining the order in which the substrategies will be applied. By this the search space of possible transformations will be kept small due to an orientation towards the goal of constructing a program. The user only has to choose parameters for the individual transformations selected by the strategy.

Together with the set of rules the design of the main strategy is crucial for the power of the resulting program synthesizer and much theoretical and experimental work has to be done to make them a good foundation for the rest of the system that ensures its correctness and completeness.

The fundamental system described so far might already be used as a program synthesizer, though a very primitive one. Good choices of the user provided, any kind of problem can be successfully transformed into a program.

Provide a metalanguage operating on top of the fundamental system that allows to write programs analyzing logic formulae and setting parameters of the transformations. It will become the entrypoint for the intelligence of the system. For instance, heuristics which instead of the user will automatically choose some of the parameters for the individual substrategies should be developed as metalanguage programs (avoiding the temptation to hack them). This allows to easily expand and modify the system according to experiences with sample syntheses.

Use a knowledge base to store domain knowledge about the underlying theory and knowledge about programming skills. Obviously, this kind of knowledge is exhaustively being used during the program development process. Thus one should not try to build a powerful program synthesizer without doing the same. Also for the final step knowledge about algorithmic contents of a formula/predicate is necessary.

Domain knowledge should by no means be implicitly encoded in the synthesizer though it is done very often and even heuristics represent knowledge that - at least to a great amount - should be abstracted and explicitly formulated. Otherwise maintainance and modification of the system will become very difficult. An explicit knowledge base also allows to let the user extend the knowledge about the specific field he will be working in. It will free the user from having to type in standard knowledge over and over again when specifying his problem (Imagine being asked what it means for a list to be ordered for the 50th time). It may, however, be meaningful to let the user certify that his notation agrees with that of the system.

If not part of the knowledge base an additional theorem prover will also improve the reasoning power of the system's heuristics.

Last but not least one must not forget user-friendliness. A powerful system which is too difficult to operate with will not be used at all. In extreme cases not even collegues of those who have built it can be convinced. Therefore a nice user interface with menus, windows, some editing power, some help mechanisms etc. should handle the interaction with the user. Unfortunately most researchers do not care about this since it is of no scientific value for them.

A system following these points will be open ended in its possibilities. While at the beginning state it may be very weak as far as automatic program synthesis is concerned it may be arbitrarily extended in its heuristics and domain knowledge. Experiences from sample syntheses will lead to the discovery of new heuristics which then will be added to the system. This can be used for both - teaching the user and increasing the power of the system which in the end will yield a powerful support mechanism for program development.

4. DESIGN OF THE LOPS SYSTEM

LOPS - abbrevation for **LO**gical **P**rogram **S**ynthesis - is an approach in the field of automatic programming which has been pursued at the Technical University Munich for several years (see [BIB 80], [BIE 85], [FRO 85]). Roughly it can be characterized as follows:

Starting from a problem specification in predicate logic a program is obtained by stepwisely rewriting the initial formula. In the course of these transformations the algorithmic content is gradually increased until a formula is obtained which in a last small step can be transformed into a program. The transformational steps are guided by a few theoretically well known strategies which contain the potential power of the LOPS approach

Like most of the systems built for practical experimentation our previous implementation of LOPS (see[BH 84]) after a while became quite bulky and difficult to maintain. Therefore decision was taken to develop a new implementation of the strategical and heuristical knowledge of the LOPS-approach.

This led to the conception and implementation of the system XPRTS, a general formula transformation system (see [NFK 87]). XPRTS is considered a sufficiently powerful basis for the flexible implementation of hopefully all kinds of strategies and heuristics and by no means tailored to the LOPS approach. In the long run, we intend to incorporate as well a great many of the ideas developed by other researchers in this field. Using this tool we have begun the reimplementation of LOPS following the guideline given in the previous chapter. Our main strategy ensures that the system is able to handle every kind of problem from the beginning. Only the degree of automation increases while we put in our experiences. Our main strategy of LOPS proceeds in 5 principal phases.

INPUT: A formula in input-language
|
v
1. Preprocessing (normalization, lemmata, subproblems ...)
|
v
2. GUESS-PHASE
|
v
3. reformulation using domain knowledge (application of lemmata)
|
v
4. GET-REC (introduce recursion)
|
v
5. Postprocessing (subproblems, "optimization" ...)
|
v
OUTPUT The computational convenient formula

The input formula will be given in a language that is a straighforward notation of first order predicate logic. Phase 1 (Preprocessing) transforms this formula into a "simpler" form that can easily be handled by following phases. Particularly normalization, application of domain knowledge (equivalence lemmata), and a subproblem mechanism belong to this phase. Phase 3 again is application of domain knowledge in order to simplify the formula before entering the next phase. Phase 5 (Postprocessing) should transform the final program into a slightly optimized form and, if necessary, put pieces generated by the subproblem mechanism together again. Phases 2 and 4 are realizations of the two fundamental strategies of LOPS which have been presented in [BIB 80]. Most of the systems capabilities will depend on the power of these two phases.

The result of this process will be also a formula in the input language. But this formula has some syntactic properties which make it easy to translate it into a "real" programming language. It may, for instance, be already quite close to the PROLOG programming language that only some control has to be generated.

According to the guideline each of the 5 phases is guided by several parameters which at the lowest level of system-automation have all to be given by the user. Since phase 3 allows user-controlled application of all kinds of equivalences the system in general is able to handle all kinds of specifications. However, it may be that the user has to provide the complete solution while the system is just checking it. By the end of the current project we expect to have implemented the complete interactive tool.

In addition to that we are building heuristics to determine the parameters and control the 5 phases on top of the interactive tool. Our open-ended hierarchy of levels of "intelligence" will allow us to increasingly automatize the program synthesis process. Thus, in the near future we expect to be able to synthesize a small class of specifications fully automated. Till now this system was able to perform syntheses of maximum and sorting problems which serve us as test examples for further development.

The system is designed in a way that it never will really be completed. There will always be a possibility to add more intelligence on top of what is already existing. Besides that, adding domain knowledge to the common knowledge base will have a similar effect. In some sense, therefore, the LOPS system will have the same learning capabilities as humans do although it is currently not planned to extend the system to perform an automatic learning feature.

CONCLUSION

First experiences with the new manifestation of LOPS show that using the concepts described in the former sections indeed lead to a promising system. However, a lot of work remains to be done. Many ideas developed in the field Program Synthesis need to be examined in practical experiments. Strategies for improving efficiency have to be collected. Heuristics have to be developed. Standard domain knowledge has to be gathered and much more.

Due to the size of the task there will be a huge amount of labour that has to be put into realizing a Programm Synthesizer before it will reach the stage of industrial usability. It is not realistic to expect just a few man-years of work to be enough. Thus for the next years Programm Synthesis will stay to be pure research and it would do the field a bad turn to claim something else.

REFERENCES

Bates,J.L. ; Constable,R.L. : *Proofs as programs.* ACM TOPLAS 7 (1), January 1985, pp.113-136.

Burstall,R.M. ; Darlington,J. : *A transformation system for developing recursive programs* JACM 24 (1977), pp. 44-67.

Bibel,W.; Hörnig,K.M. : *LOPS - a system based on a strategical approach to program synthesis.* Automatic program construction techniques, (A.Biermann, G.Guiho, Y.Kodratoff, eds.), MacMillan, New York 1984, pp. 69-89.

Bibel,W. : *Syntax-directed, semantics-supported program synthesis.* Artificial Intelligence 14 (1980), pp. 243-261.

Biermann,A.W. : *Some Examples of Program Synthesis.* Automatic Program Construction Techniques (A.W. Biermann, G. Guiho, Y. Kodratoff ed's.), Macmillan Publishing Company, London, 1984, pp. 553-562.

CIP Language Group : *The Munich project CIP, Vol.I: The wide spectrum language CIP-L.* Lecture Notes in Computer Science Vol 183, Springer, Berlin, 1985.

Dershowitz,N. : *Synthetic Programming.* Artificial Intelligence 25, 1985, pp.323-373.

Franova,M. : *Program Synthesis and Constructive proofs Obtained by Beth's tableaux.* Cybernetics and System Research 2 (R. Trappl ed.), North-Holland, Amsterdam 1984, pp. 715-720.

Fronhöfer,B. : *The LOPS-Approach: Towards New Syntheses of Algorithms.* ÖGAI-85, Vienna, Austria, September 1985, (H.Trost, J.Retti, eds.), Informatik-Fachberichte 106, Springer, Berlin 1985, pp. 164-172.

Hogger,C.J. : *Derivation of Logic Programs.* Journal ACM, Vol.28, 1981, pp.372-392

Manna,Z. ; Waldinger,R. : *The automatic synthesis of recursive programs.* Proc. ACM Symposium on AI and Prog. languages, Rochester (1977), pp. 29-36.

Manna,Z. ; Waldinger,R. : *Synthesis: dreams => programs.* IEEE Transactions of Software Engineering SE-5, (4), 1979, pp.294-328.

Manna,Z. ; Waldinger,R. : *A deductive approach to program synthesis.* ACM TOPLAS 2, (1), 1980, pp.90-121.

Neugebauer,G.; Fronhöfer,B.; Kreitz, C. : *XPRTS - An implementation tool for program synthesis* Report ATP-80-X-87, Forschungsberichte Künstliche Intelligenz, Technical University of Munich, 1987.