XPRTS
An Implementation Tool for Program Synthesis

Gerd Neugebauer 2
Bertram Fronhöfer 3
Christoph Kreitz 2

Abstract

The paper presents first results of the reimplementation of LOPS, an approach in the field of program synthesis. To this end a general development system, called XPRTS, has been designed and implemented. We will describe its architecture and programming model. On this basis, the reconstruction of LOPS has already yielded first results. The leading idea is an implementation of strategies and heuristics on several layers of abstraction. A glance at this implementation work also shows how XPRTS is used.

Keywords: Program synthesis, Program Construction, Program Transformation, Logic Programming

1 Introduction

We consider the goal of Program Synthesis to be the highly automatic construction of efficient programs from complete specifications of the relationship between input and output data.

To this end a new implementation of the strategical and heuristical knowledge of the LOPS-approach is under way. LOPS is an approach to program synthesis which has been developed at the Technical University of Munich; see [Bib 80] or [Fro 85] for theoretical issues and [Bib 84] for past implementation work. The current reimpiementation is done on the basis of a general program development system (XPRTS).

This work is pursued within the ESPRIT project No. 973 ALPES which aims at an Advanced Logical Programming Environment.

The importance of a Program Synthesis tool for a Logic Programming Environment is a twofold one: It would liberate the programmer from restrictions inherent to current Logic Programming languages, e.g. the limitation to Horn clauses, and the programmer would be relieved from efficiency considerations:

(a) The natural definition of many programming problems is of a form that is different from Horn clauses, which constitute the basis of current Logic Programming languages. A typical example is the subset relation which is naturally specified by a universally quantified implication. The use of this definition in programming would be possible in a Logic Programming Environment in which a Program Synthesis tool bridges the gap to the particular respective Logic Programming language.

(b) It is certainly desirable that the programmer could devote all his efforts to the analysis of the problems he has to tackle, and not be distracted by efficiency considerations: The programmer should only concentrate on the correct specification of his problems and leave it to the system to find an efficient algorithm, or an executable algorithm for that matter.

1 This work has been funded by ESPRIT and DFG
2 FG Intellektik, FB Informatik, Technische Hochschule Darmstadt, Alexanderstr. 10, D-6100 Darmstadt
3 Institut für Informatik, Technische Universität München, Arcisstr. 21, Postfach 202420, D-8000 München 2
The Program Synthesis tool we are implementing at present will perform syntheses as far as possible in an automatic way. Although it is in sight, that in the near future, complete automation of Program Synthesis will be feasible for a large class of problems, nevertheless there will remain specifications that our system will hardly be able to cope with. Therefore the user must be able to observe and influence the synthesis processes via an elaborate user interface. This component also provides indispensable assistance in the further development and refinement of the Program Synthesis system.

Despite its long history, Program Synthesis must still be considered a field of basic research. Although some fundamental techniques are well-known, the main problem nowadays is the conception of strategies and heuristics which guide these basic mechanisms of a Program Synthesis System.

This kind of knowledge cannot be developed purely theoretically, but requires practical experimentation. Unfortunately, systems built and used for this purpose, after some time tend to get quite bulky and offer great maintenance problems. This was also the case with the old LOPS implementation described in [Bib 84] and therefore we decided to develop a new system.

This lead to the conception and implementation of the system XPRTS, a general formula transformation system. XPRTS is by no means tailored to the LOPS approach. On the contrary, we consider it a sufficiently powerful basis for the flexible implementation of hopefully all kinds of strategies and heuristics. In the long run, we intend to incorporate as well a great many of the ideas developed by other researchers in this field.

The plan of the paper is the following:

In section 2 we will dive into XPRTS, the innermost part of our program synthesis system and will outline its architecture.

In section 3 we will give a brief introduction to the LOPS approach, which is both our tradition and the guide line of our present implementation work, and

In section 4 we will show the results which we have achieved up to now in the course of our reimplementation of LOPS.

2 The XPRTS System

2.1 XPRTS — A General View

As already mentioned in the introduction, past experience suggested a redesign of the LOPS system. The essence of the LOPS approach is its strategical framework which constitutes a general methodology for Program Synthesis. In our old prototype these strategies were coded in a rather direct way.

In contrast to our old work, we now started by developing a general tool. This general tool is called XPRTS — eXperimental PRogram Transformation System. Due to the logic orientation of LOPS, XPRTS is intended to provide a general tool for formula transformation and manipulation. This open design shall also enable us to implement as well any other kind of heuristics concerning logic based program synthesis and serve as a tool to reconstruct and experiment with different program synthesis attempts.

The formulae XPRTS can cope with are restricted to first order predicate logic. This language is employed to state the synthesis problem and the necessary knowledge.

It is also used to express all intermediate results of the Program Synthesis process — i.e. equivalent descriptions of the specified problem — among them also the rather final result which is a “program” in Horn clauses. This program might not run with a given PROLOG interpreter, because the fixed PROLOG control might lead the program into an infinite branch which has no solution, whereas the existing solution is not found. Therefore, a postprocessing step transforms the Horn clause program into a PROLOG program. This is the only step in the synthesis process, where knowledge about the used programming language is needed. Here we have to include the non-logical features of the target PROLOG
language. Among them are the control constructs — implicit or explicit (cut) — and (logically) incorrect implemented build-in predicates.

To allow reasonable use of the XPRTS-system for experimentation it is designed as an interactive one. However, this does not mean that the user is forced to do everything himself. Instead, the experienced user is encouraged to view and control the system's actions. The novice, on the other hand, is guided by the system to achieve the results he is aiming at. As an extreme case, this guidance can be the completely automatic synthesis of a program.

2.2 XPRTS — The Architecture

![figure 1: The components of XPRTS]

XPRTS consists of several independent modules and a hierarchy of control units representing various levels of abstraction on which the syntheses may be carried out. For the architecture see figure 1.

Mainly we can distinguish three kinds of modules. First, the kernel which consists of the transformation unit TU, the verification unit VU and various control units CU. Second, the external modules which are not implemented within the scope of our current activities. These are an editor, a theorem prover, a data base management system. Concerning these parts we may profit from the work of other projects going on in our research group in Munich. Third, the communication unit ComU which offers a comfortable user interface.

2.2.1 The Transformation Unit

The transformation unit (TU) is designed to transform formulae. To do so it takes as arguments the description of the desired transformation and returns its result and an error code indicating the success or failure.

The transformation unit provides only very simple actions like manipulation routines for trees — as formulae are internally represented — (copying, truncation, expansion, comparison).
2.2.2 The Control Units

The transformation unit is only able to apply basic transformations to a program. This corresponds to the lowest level of abstraction.

The most primitive control unit (CU₀) contains a few basic control constructs. These constructs allow the combination of the basic rewrite actions of the transformation unit into more complex procedures on which the higher system layers work, e.g. definition of (recursive) procedures, or conditional processing of actions.

More elaborate control units (CUᵢ, i > 0) reflect higher levels of abstraction of strategic knowledge. Each of them is based on the respective lower ones. This implies that on higher levels the transformations are specified by means of the utilities which are provided by the lower levels. The higher the level the less it is reflected in what way the programs are really represented or how the transformations are actually carried out. This process of abstraction will be completed until the heuristical ideas we want to include can be expressed in a natural and simple way.

Concerning the development of intermediate levels, we think that these will provide us with very general and powerful tools, so that nearly every sort of heuristics guided program synthesis system can be realized within this framework.

2.2.3 The Verification Unit

During the synthesis process, strategies may suggest formulae, e.g. equivalences, which must then be approved before they can be used for the synthesis. The verification unit (VU) tries to verify such a formula within a given time limit. To do so it first looks in the knowledge base to see whether the formula is already there. If this is not the case, the VU gives the problem to a theorem prover together with a time limit. Finally, if nothing succeeds it may eventually ask the user to accept or deny the theorem.

2.2.4 The Knowledge Base

The knowledge base interface is a very important part of the program synthesis system. Our knowledge base is structured according to several criteria to enable an efficient access to the stored information. There are several kinds of information which must be separated in a natural way. Among these is the programming knowledge, i.e. knowledge about general strategies in programming, knowledge about the chosen target language and knowledge about general statements which are true and could be useful.

3 A Summary of LOPS

Before we present our current implementation work of the LOPS approach, we want to give a brief summary of the principal features of this approach.

LOPS — abbreviation for LOgic Program Synthesis — is an approach in the field of automatic program construction. The starting point of the program synthesis is a specification given as a formula in first order logic. A series of correctness-preserving transformations which are guided by strategies and supported by deductive tools leads to an algorithmically "better" formula which can easily be transformed into a program.

A successful LOPS synthesis can be separated into two major parts, which are centered around the strategies GUESS-DOMAIN and GET-REC. To apply these strategies, some preprocessing and optional postprocessing steps may be necessary in both cases.

The GUESS-DOMAIN strategy tries to find an appropriate portion of the specification which can be used to split the input into smaller pieces and compute the desired output recursively from these pieces.
Depending on the results of the GUESS-DOMAIN strategy, the strategy GET-REC tries to introduce recursion. According to the chosen recursion scheme, the input may be split in different ways.

After recursion is introduced, a part of the formula may be found to be untreated. This part has to be made directly evaluable or it can be treated with the same mechanism again. The result of this process is a formula which can be easily translated into a target programming language. Note that these strategies are designed to leave some choices open which could be made by an assisting user. So they are suitable to cover a wide field of possible syntheses.

A detailed description of the approach and the strategies can be found in [Bib 80] and [Bib 84]. More recent theoretical results which further develop this approach are [Fro 85] and [Neu 86].

Let us illustrate the strategies just presented by a small standard example, the synthesis of an algorithm for calculating the maximal element of a finite set of numbers.

The specification we start from consists of the definition of a predicate (i.e. an input-output relation) together with a mode (i.e. an indication about which of the variables of the predicate are input variables and which are output variables). We consider the following definition:

\[
\text{max}(S, m) \iff m \in S \land S \leq m
\]

The mode for which we consider this program specification is that \( S \) is input variable while \( m \) is output variable.

The strategy GUESS-DOMAIN introduces a guess variable \( g \), together with the specification of a domain where the guess should be carried out. In the case of our example the domain specification \((DS)\) is based on the subformula \( m \in S \). We define:

\[
DS(S, g) \overset{\text{def}}{=} g \in S
\]

Since guessing can be right or wrong, these two possibilities are expressed by the disjunction \( g = m \lor g \neq m \), which is added to the formula. All this leads to the following transformation of the initial problem specification:

\[
\text{max}(S, m) \iff g \in S \land
\]

\[
[(m \in S \land S \leq m \land g \neq m) \lor (m \in S \land S \leq m \land g = m)]
\]

The strategy GET-REC now introduces recursion through substitution of \( S \) by \( S \setminus g \) in the failure case. During this step the formula \( g \neq m \) which distinguishes the failure case is transformed as well (see [Bib 80] or [Fro 84] for details).

Finally, through application of folding with the initial specification we obtain:

\[
\text{max}(S, m) \iff g \in S \land
\]

\[
[(\text{max}(S \setminus g, m) \land m > g) \lor (m \in S \land S \leq m \land g = m)]
\]

What remains to be done is the generation of control for the execution of this recursive program as well as further simplifications, which take into account the order of execution. This leads to:

\[
\text{max}(S, m) \iff g \in S \land
\]

\[
[(\text{max}(S \setminus g, m) \land m > g) \lor g = m]
\]

which is already close to a PROLOG program which reads:

\[
\text{max}(S, M) :- \text{in}(G, S), \text{max}1(S, M, G), !.
\]

\[
\text{max}1(S, M, G) :- \text{set.minus}(S, G, S1), \text{max}(S1, M), \text{greater}(M, G).
\]

\[
\text{max}1(S, G, G) .
\]

To get an executable PROLOG program appropriate definitions for the predicates in, set.minus and greater have to be provided by the system or the user.
4 The LOPS Implementation on XPRTS

Having presented in the last two sections the architecture of XPRTS and the basic principles of LOPS, we will show now in an exemplary way how the latter are implemented on XPRTS.

The syntactic mechanism behind LOPS is the application of equivalence transformations to a problem specification until an executable (and, let us hope efficient,) program is reached. Such a specification, as well as the derived intermediate problem descriptions, are of the form:

\[ \text{precondition} \rightarrow (\text{head} \leftrightarrow \text{body}) \]

Usually all the transformations operate only on the body which is assumed to be kept in DNF (disjunctive normal form) as far as is reasonable. The precondition is only necessary for validity checks — note that it was missing in the maximum example — and the head represents the name of the problem/program itself.

If we look at the architecture of XPRTS several Control Units can be distinguished. Each of them corresponds to a level of strategical and heuristical knowledge. Up to now we implemented the bottommost levels 0 and 1, which implement basic operations and elementary transformations. Work on level 2 is still in an experimental phase.

We will give a rough glance at these levels and their contents, and conclude with an exemplary discussion of how a particular strategy is distributed over the different levels.

**Level 0: Basic operations**

Level 0 comprises operations that are necessary for the analysis and synthesis of formulae from a rather logical point of view. For this reason a collection of constructors and destructors for terms and formulae has been implemented and also some elementary list operations. In addition to that, the purely transformational ingredients of the LOPS strategies belong to level 0. Examples are syntactic procedures like matching, substitution, substructure descriptions etc.

**Level 1: Elementary transformations**

At level 1 LOPS should be able to perform elementary formula transformation on a higher logical level. It should become less interesting than on level 0 how formulae are handled internally in the XPRTS machine. Syntactic procedures like matching, substitution, substructure descriptions etc. will more and more become background commands or control/analysis parts of the metalanguage. That is, they still may be used for analyzing formulae. The only commands which actually do formula transformation, however, are the ones that will be described here.

Transformations at level 1 still do not contain heuristic components. They are the basic tools which are to be controlled by a strategy at some higher level. Thus commands mentioned here should have parameters to be controlled by either the user using the system on this level or the heuristics. We try to split the strategies occurring in the previous LOPS system into these major parts. The advantage will be that we first can gather some experience with formula transformation/program synthesis “by hand” before we actually go on to programming the strategic part of the system. For better control every transformation can also operate on only a certain part of a formula.

We will simply give some examples of transformations of this level:

- Equivalence transformations given by some lemma (TRANSFORM_BY):

  A lemma of the form

  \[ \text{precondition} \rightarrow (\text{head} \leftrightarrow \text{tail}) \]

  shall be applied to a formula in either forward (\text{head} \rightarrow \text{tail}) or backward direction. That is, some occurrences of \text{head} will be replaced by \text{tail} provided \text{precondition} is satisfied or vice versa.

  This is the most important transformation for the implementation of LOPS, as well as of a program synthesis system in general, since in general every strategy uses equivalence transformations. For
efficiency reasons, however, some strategies (like GUESS) use a more straightforward implementation rather than referring to TRANSFORM_BYP.

The ground version of this transformation is that head is just a “simple” literal matching some subterm of the formula’s body. In that case we have to instantiate the precondition according to the matching-list and check whether it is satisfied in the immediate neighbourhood of the subterm. Then the head can simply be substituted by the instantiated version of the tail.

Extensions of this transformation would be e.g. that conjunctions of literals must be matched against conjunctions of literals. Here associativity and commutativity must be taken into account.

Special versions of this transformation are the well-known fold and unfold operations.

- Elementary guessing
  (with parameters under higher-level control: BASIC.GUESS)

  BASIC.GUESS will transform a formula of the form

  \[
  \text{precondition}(i_1, \ldots, i_n) \rightarrow (\ SP(i_1, \ldots, i_n, y_1, \ldots, y_m) \leftarrow OC(i_1, \ldots, i_n, y_1, \ldots, y_m))
  \]

  where \( OC \) is a formula in clausal form and \( SP \) is the name of the problem to be synthesized, into

  \[
  \text{precondition}(i_1, \ldots, i_n) \rightarrow (\ SP(i_1, \ldots, y_1, \ldots) \leftarrow OC(i_1, \ldots, y_1, \ldots) \land \text{domain} - \text{condition}(g) \land (\text{taut}(y_j, g) \lor \neg\text{taut}(y_j, g)))
  \]

  where \( \text{domain} - \text{condition}(g) \) is a proper subset of the conjuncts in \( OC(i_1, \ldots, y_1, \ldots) \) with \( y_j \) replaced by \( g \) and \( \text{taut} \) is some tautology predicate which conveys some relation between the output-variable \( y_j \) and the guess-variable \( g \).

  Control parameters are descriptions about how to select the subset of \( OC \) and the output-variable respectively and the tautology predicate (equality chosen as standard).

  Further examples are primitive versions of LOPS strategies (the more elaborate versions of which will be met on level 2). The difference between the two levels is that on level 1 the transformations are heavily parameterized while on level 2 these parameters will, at least partially, be calculated by heuristics (cf. example at the end of section 5). In some way level 2 can be envisioned as a kind of preprocessing that is necessary before the more primitive transformations of level 1 can be called.

  Now it will not be a surprise, to a reader familiar with the strategies of LOPS, that on level 1 transformations are found, the names of which are BASIC.REC, BASIC.EP, and BASIC.RNV, etc.

  These transformations are primitive versions of the respective level 2 strategies GET-REC, GET-EP, and GET-RNV.

  Level 2: Level 2 and higher levels are still in an experimental state: They should finally include all the strategies from [Bib 80]

  - A higher version of the GUESS-strategy

    It will use the I/O-Graph method (see [Neu 87]) to select an appropriate output variable. Different tautology predicates found in the knowledge base may be tried and semantic knowledge on the predicates involved will be applied in order to find the domain specification. This requires a lot of knowledge to be stored in the knowledge base.

    Additionally, the following strategies are being implemented on this level

    - GET-REC which depending on knowledge about recursion schemes is responsible for the introduction of recursive calls into the programs.

    - GET-RNV reduces the number of output variables by trying to express one output variable in terms of the others.
GET-SOC tries to split the output condition into parts where exactly one of the output variables occurs. In this case program synthesis would first result in a collection of single algorithms which have to be combined in the end.

**TOP-level: LOPS**

The TOP-level represents LOPS as the user will see it. The user will only call the command LOPS and everything else will be controlled by the system. Single strategies used within it (like GUESS, GET-REC etc.) will be implemented as TOP-level commands calling strategies from the highest implemented level. This allows to improve the system without changing it's general structure.

Finally, let us show by means of the maximum example how the strategy GUESS is now realized in our implementation.

Here we can only present basic versions of guess which belong to level 1. We want to give an impression of how the raising of levels of abstraction is reflected in the implementation. For sake of readability and brevity we have omitted some very low procedures and concentrate on two guessing variants presented in a bottom-up way.

```prolog
% This procedure determines the parameters for guess
% :define ( basic_guess_sd
( $FORMULA, % formula to apply guessing to
 $SD, % description of subformula
 $DCLIST, % description of domain
 $GUESS_VAR, % description of guess variable
 $TAUT_PRED % tautology predicate to be used
)
{ :let ($DOHAIN, %--- extract the domain from formula
 :getl (:get(:body($FORMULA),$SD), $DCLIST))
 { :let ($NEW_VAR, %--- create a new variable for guessing
 :mkvar ($GUESS_VAR))
 { :let ($TPRED, %--- instantiate tautology predicate
 :make_literal ($TAUT_PRED, [$GUESS_VAR,$NEW_VAR]))
 { :let ($DC, %--- instantiate domain
 :subst([[[$GUESS_VAR,$NEW_VAR]],$DOHAIN)])
 { %--- collect the pieces
 :t_guess_sd ($FORMULA,$SD,$DC,$TPRED).
 }.}.}.}.}.

This procedure is based on t_guess_sd. This procedure directly calls level 0 system primitives which really transform the formula $FORMULA. This version still has an argument, namely $SD, which describes exactly a subformula which is used as a first restriction for the domain specification to be chosen.

In a next version (see below) this subformula description is specialized to the whole formula. Additionally, another heuristic is used to determine the predicate $TAUT_PRED to construct a tautological case differentiation. This heuristic is implemented by the procedure find_taut_pred. Here we see that some parameters of the lower level procedure are instantiated by the one higher up.
```

This procedure is based on t_guess_sd. This procedure directly calls level 0 system primitives which really transform the formula $FORMULA. This version still has an argument, namely $SD, which describes exactly a subformula which is used as a first restriction for the domain specification to be chosen.

In a next version (see below) this subformula description is specialized to the whole formula. Additionally, another heuristic is used to determine the predicate $TAUT_PRED to construct a tautological case differentiation. This heuristic is implemented by the procedure find_taut_pred. Here we see that some parameters of the lower level procedure are instantiated by the one higher up.
\%--- find a tautology predicate
:let ( \$TAUT_PRED, :find_taut_pred($FORMULA,$DCLIST,$GUESS_VAR) )
\%--- apply the guess transformation
:basic_guess_sd ($FORMULA,[ ],$DCLIST,$GUESS_VAR,$TAUT_PRED).
).

The task of the higher order guessing procedures is to fill up the remaining arguments, e.g. the output variable for which we do guessing. This information is necessary to determine the tautology predicate \$TAUT_PRED. Here the full power of our heuristic ideas is used. Thus the corresponding procedures are much larger than the presented ones.

Example:

We come back to our previous maximum example and show how the parameters are instantiated when the above procedures for the GUESS strategy are called during the synthesis. From a higher level \std_guess is called, namely

:std_guess( max(S,M) <-> ( in(M,S) & le(S,M) ) , [[1]], M ).

The first argument is the problem specification, the second one is the description of a subformula which refers to \( \text{in}(M,S) \), and the third argument contains the variable for which we want to do guessing.

During the execution of this call to \std_guess the procedure \find_taut_pred yields the equality predicate eq. This results in the following call to \basic_guess_sd

:basic_guess_sd ( max(S,M) <-> ( in(M,S) & le(S,M) ) , [ ] , [[1]], M, eq ).

during the execution of this call various intermediate results are computed and we arrive at the call

:t_guess_sd ( max(S,M) <-> ( in(M,S) & le(S,M) ) , [ ] , in(g,S), M, eq ).

After all of these computations we obtain the formula

\[
\text{max}(S,M) \leftrightarrow \text{in}(g,S) \& ( \text{in}(M,S) \& \text{le}(S,M) ) \& ( g=m \text{ OR } g\neq m ).
\]

which is the input for the further synthesis steps.

5 Conclusion

In this paper we presented current work on a program synthesis system.

In contrast to other work in Program Synthesis this system is different in the following respects:

- many systems are mainly prototypes or direct implementations of a single idea for Program Synthesis. (Among these are the old LOPS system or the system PRECOMAS [Fra 86] which was developed in the same ESPRIT project) However, our system is designed to provide an open experimenal basis for logical program construction.

- although strongly logic-oriented, XPRTS doesn't follow the approach to Program Synthesis via constructive proofs, as proposed by [Man 80], but works in a more transformational way as advocated in [Hog 81].

- finally, the orientation to logic and logic programming is very different from a lot of past work in program transformation, where functional and procedural languages were handled, as for instance [Bur 75] or [Bau 82].
Among all other approaches the philosophy of KIDS [Smi 88] is closest to ours. It uses a similar architecture based on a commercial knowledge-based programming environment.

The conceptual basis of our system is the hierarchical structure of abstraction levels which is provided by the XPRTS system. Lower level operations will be operations which transform formulae according to parameters handed down from the higher levels, where they are determined by more and more general heuristics.

The LOPS program synthesis strategies will be implemented on top of the general tool XPRTS. This development has already been started and is yielding first results. Due to the clear structure of XPRTS, we believe that on this basis some past theoretical results on LOPS — as for instance syntheses for problems like a minimal cost spanning tree (Kruskal's algorithm) or linear pattern matching — can be implemented. (Their incorporation into the old LOPS implementation seemed not to be feasible.)

As our long term research goal, we intend to use this system for future experimentation, including as well research results published by other researchers. We expect this experimentation to lead to new insights into program synthesis which eventually will result in the implementation of still more powerful strategies on higher levels.

Acknowledgements

We thank Ulrich Furbach for contributing valuable comments to this paper.

References


