Automatic program synthesis via synthesis of loop-free segments*

by JOE W. DURAN

The University of Texas at Dallas
Richardson, Texas

INTRODUCTION

Work on theorem-proving-based automatic program synthesis (see Lee, et al.,1 for example) has been neglected lately. In their 1971 paper, Manna and Waldinger* covered one of the main reasons why—the difficulty of synthesizing program loops within the current state of the art of automatic theorem-proving. However, there is a great deal of continuing work in theorem-proving, and it is important that motivating work in related areas such as program synthesis not be neglected.

Most theorem-proving-based synthesis systems attempt to constructively prove theorems of the form, “for all input values satisfying the desired input predicate I, there exists a corresponding set of values of output variables which satisfy the desired result predicate, R.” The resulting program (if any) is correct with respect to the input and output predicates. In general, inductive proofs are needed to synthesize the loops within a program. This causes difficulty, and in fact previous work has not been notably successful with loops. To quote Manna and Waldinger, “... these systems have been fairly limited; for example, either they have been completely unable to produce programs with loops or they have introduced loops by underhanded methods.” Since most interesting programs contain loops in some form, this is a crucial problem for successful synthesis. We will limit our discussion to iterative loops.

Manna and Waldinger outlined an approach to automatic synthesis which attacked the problem of iterative loops, and discussed the use of various forms of induction for reducing synthesis to the proving of loop-free (i.e., induction-free) lemmas. However, they were dissatisfied with the large number of equivalent induction principles required by their approach. In fact most interesting programs contain loops in some form, this is a crucial problem for successful synthesis. We will limit our discussion to iterative loops.

Let us consider some decomposition of our function $F_p$, such that

$$F_{n}(F_{n-1}(\ldots(v_{in})\ldots))=F_{p}(v_{in}) \tag{2}$$

Let $P_{j}$ be a program for computing $F_{j}$. A program computing $F_{p}$ may be constructed by concatenating the programs $P_{1}, \ldots, P_{n}$. Though the decomposition (2) may be trivial, for most programs there will exist non-trivial ones.**

Any concatenation $P_{1}P_{2}\ldots P_{n}$ of $P_{j}$s satisfying these

** It will be recognized that this decomposition is included in the very well known concept of stepwise refinement in program construction (see Mills*).
predicates will produce a program which computes \( F_R \), provided that each \( P_j \) is constructed to terminate. The program so produced will thus be correct with respect to \( I \) and \( R \).

We proceed to show that, under this scheme, there will always be a satisfactory set \( \{ P_1, P_2, \ldots, P_s \} \) such that each \( P_j \) contains at most one loop. Since any computable function can be computed by a program with a single loop, each \( F_j \) can be computed with at most a single loop. Each such single-loop program has an associated loop invariant which is sufficient for a Floyd-type correctness proof. This follows from the loop invariant existence proofs in References 2 and 4. Therefore there will always be a satisfactory set of \( P_j \)'s such that each one contains at most one loop, which can always be of the form ‘(initialization) while \( B \) do (loop body)’. \( F_R \) can thus be computed by a concatenation of single loop program segments which have associated loop invariants. We will use this fact to complete the analysis leading to our characterization of general program synthesis in terms of loop-free lemmas.

Manna and Waldinger considered synthesis in terms of the proof of first-order theorems of the form

\[
\forall v_{in} [I(v_{in}) \to (\exists v_{out}) R(v_{in}, v_{out})],
\]

where \( v_{in} \) is the vector of input variable values, and \( v_{out} \) the vector of output variable values. Let us now consider any arbitrary single loop program with while-do loop form, correct with respect to input and result predicates \( I \) and \( R \). Such a program has two segments—the initialization code and the loop body code—and has a loop control predicate, \( B \), and a loop invariant, \( Q \), such that \( Q \land \neg B \Rightarrow R \). We can synthesize the two segments separately and then use the two segments to assemble the program into its while-do structure.

Initialization code must produce output satisfying the loop invariant whenever the input predicate is true. Thus the loop invariant acts as the result predicate of (3) and initialization can be stated in terms of the induction free lemma

\[
\forall v_{in} [I(v_{in}) \to \exists v_{0} [Q(v_{0}, v_{0})]],
\]

where \( v \) is the vector of all program variables, with \( v_0 \) representing its initial value.

The input conditions for loop body code entry are that the loop control predicate, \( B \), is true and that the loop invariant, \( Q \), is true. The desired output of the loop body code must satisfy the loop invariant. Further, since we wish to synthesize programs which terminate, we require that the input to the loop body code, which satisfies \( B \), must be transformed into output which is closer to not satisfying \( B \). The predicate ensuring this, denoted by \( P(v, v') \), is essentially Dijkstra’s variant function.9 (Here \( v \) is the program variable vector before loop execution and \( v' \) the updated value after execution.) The complete result predicate is thus \( (Q \land P) \) and the loop body code lemma is stated as

\[
\forall v_{in} \forall v [B(v) \land Q(v_{0}, v_{0}) \to \exists v' [Q(v_{0}, v') \land P(v, v')]],
\]

SYNTHESIS

Existing first order synthesis systems can be applied to the above lemmas after they are generated for a given problem. For instance, Lee, *et al.*, have presented a resolution based algorithm for producing loop-free programs which is proved to generate correct programs. Their system constructively proves theorems of the form

\[
\forall v_{in} \forall v_{out} \{R(v_{in}, v_{out}) \to A_NS(v_{out})\}.
\]

The predicate \( A_NS(v_{out}) \) is used to record the unification substitutions which take place for \( v_{out} \) during a resolution proof. To use their methods, lemmas 4 and 5 are converted to the form of (6). Our desired result predicate for initialization is \( Q \), and for loop body code is \( Q \land P \). Although Lee, *et al.*, make no explicit mention of input conditions, clearly these can be included in their form. Thus (6) can be used for initialization by letting \( R = \{B \land Q(v_{in}, v) \land P(v, v')\} \), giving us

\[
\forall v_{in} \forall v_{out} \{I(v_{in}) \land Q(v_{in}, v_{0}) \to A_NS(v_{0})\}
\]

and

\[
\forall v_{in} \forall v \forall v' \{B(v) \land Q(v_{in}, v) \land Q(v_{in}, v') \land P(v, v') \to A_NS(v')\}.
\]

If the assumptions (or axioms) that \( I(v_{in}) \), \( B(v) \), and \( Q(v_{in}, v) \) are true are added, (7) and (8) can be shown to be logically equivalent to (4) and (5). Thus we have logical equivalence where \( I \), \( B \), and \( Q \) are true, which is the only real case of interest.

The following example illustrates our approach in conjunction with resolution based loop-free synthesis as in Reference 7.

**Example 1: Factorial Function**

\[
I = \{N \geq 0\} \quad R = \{z = f(N)\}
\]

\( f \) is the factorial function, defined by \( f(0) = 1 \) and \( f(x+1) = (x+1) \cdot f(x) \). We choose \( Q(z, k) = \{z = f(k)\}, \) ~\( B(k, N) = \{k = N\}, \) and \( P(k, k') = \{k < k'\} \), so that \( Q \land \neg B \Rightarrow R \). (The non-active elements in \( v_{in} \) and \( v \) are suppressed in the above argument lists.)

- **Initialization code**—From (7), the theorem to be proved is \( \forall N \forall \forall k(I(N) \land Q(z, k) \land A_NS(z, k)) \), or \( \forall N \forall \forall k(N \geq 0 \land f(z) = f(k) \Rightarrow A_NS(z, k)) \).

From \( f(0) = 1 \), we have \( Q(1, 0) \). Expressing the theorem in clause form appropriate for performing resolution, we have

\[
(a) \quad \neg I(N) \lor \neg Q(z, k) \lor A_NS(z, k).
\]

The axioms specifying \( I \) and \( Q \), as required for resolution,
are

(b) \( I(N) \) (since we are interested only in cases where \( I \) is true) and

(c) \( Q(1,0) \) (by definition of factorial).

Resolving clauses (a) and (b) yields

\[ \neg Q(z,k) \lor \text{ANS}(z,k). \]

Resolving clauses (c) and (d) yields

\[ \text{ANS}(1,0). \]

The unification substitutions, carried in \( \text{ANS} \), are \( 1 \to z \) and \( 0 \to k \). From this, initialization code is \( z:=1; k:=0 \).

- **Loop body code**—From (8) the loop code generation theorem may be stated as the clause

\[ \neg B(k,N) \lor \neg Q(z,k) \lor \neg Q(z',k') \lor \neg P(k,k') \lor \text{ANS}(z',k'). \]

From the definition of factorial, we have the axiom

\[ \neg Q(z,k) \lor Q(z*(k+1),k+1). \]

From the hypothesis that control is still in the loop, since that is our condition of interest, we have

\[ (c) \quad B(k,N), \text{which is the condition,} \quad "B \text{ is true}" \quad \text{and} \quad (d) \quad Q(z,k), \text{which means,} \quad "\text{the loop invariant holds}." \]

The axioms needed to specify the progress requirement predicate, \( P=\{k \leq k'\} \), are expressed by the clauses

(e) \( \neg P(0,L) \lor P(M,M+L) \)

(f) \( P(0,1) \).

A string of resolutions leading to isolation of \( \text{ANS}(z',k') \) is as follows:

\[ \text{(g)} \quad Q(z*(k+1),k+1), \text{from (b) and (d)} \]

\[ \text{(h)} \quad \neg B(k,N) \lor \neg Q(z,k) \lor \neg P(k,k+1) \lor \text{ANS}(z*(k+1),k+1), \text{from (a) and (g)} \]

\[ \text{(i)} \quad \neg Q(z,k) \lor \neg P(k,k+1) \lor \text{ANS}(z*(k+1),k+1), \text{from (c) and (h)} \]

\[ \text{(j)} \quad \neg P(0,1) \lor \text{ANS}(z*(k+1),k+1), \text{from (d) and (i)} \]

\[ \text{(k)} \quad \neg P(0,1) \lor \text{ANS}(z*(k+1),k+1), \text{from (e) and (j)} \]

\[ \text{(l)} \quad \text{ANS}(z*(k+1),k+1) \text{ from (f) and (k)} \]

Thus the loop body code, derived from the substitutions for \( \text{ANS} \) arguments leading to (i), is \( z:=z*(k+1); k:=k+1 \).

Example 1 illustrates a difficulty with resolution-based methods—the fact that rather oblique axiomatization is necessary to define \( Q \) and \( P \).

It is desirable to have an automatic (or semiautomatic) synthesis system which parallels good programming methodology. Thus we might wish to carry out the necessary theorem proving by the use of "natural deduction" rather than resolution. An interactive approach, allowing human intervention, is the best immediate hope for a practical synthesis system. Natural deduction systems, by definition, retain theorems and intermediate steps in a form near to that which humans usually use. The deductions are kept reasonably close to the rules of inference usually used by humans. Additionally, such systems have much of the necessary axiomatization built into their deductions, definitions, and rewrite rules. This greatly simplifies setting up a synthesis problem, compared to resolution. Of course, natural deduction systems are usually incomplete, but this is not expected to be a practical handicap. Bledsoe and Brucki\(^{\text{b}}\) have presented a system, called PROVER, which combines natural-deduction-like theorem proving with a capability for man-machine interaction. PROVER has been modified for use in a practical correctness proving system (Good, London and Bledsoe\(^{\text{b}}\)). Much of this adaptation should be useful also for synthesis proofs.

The lemma schemata (4) and (5) are already in natural form. Example 2 shows the form applied to the integer multiplication problem and illustrates a natural deduction style proof, after the methods of PROVER.

**Example 2: Integer Multiplication**

\[
I = \{ y \geq 0 \land z \geq 0 \} \\
R = \{ x = y \cdot z \} \\
Q = \{ x + c \cdot d = y \cdot z \} \\
\neg B = \{ c = 0 \}
\]

- The initialization theorem is

\[ (i) \quad \forall y \forall z \{ y \geq 0 \land z \geq 0 \to \exists x \exists c \exists d \{ x + c \cdot d = y \cdot z \} \}
\]

The quantifiers are removed by treating universally quantified variables as constants, identified by the subscript \( o \), as in \( y_o \). Existentially quantified variables are left as is.

Thus we have

\[ (ii) \quad y_o \geq 0 \land z_o \geq 0 \to x + c \cdot d = y_o \cdot z_o . \]

The \( o \) subscript indicates that no value substitutions can be made. Since \( y_o \) and \( z_o \) are general constants, any substitution for \( x, c, d \) which satisfies (ii) will satisfy (i). Thus, for our purposes, (i) and (ii) are logically equivalent. In the PROVER system, an hypothesis of the form in (ii) is removed and added to the axiom list, leaving the basic theorem to be proved as

\[ (iii) \quad x + c \cdot d = y_o \cdot z_o . \]

A key problem for synthesis proofs is existential inference, as necessary to satisfy (iii). A natural deduction system will certainly require heuristic methods for guessing these inferences, which can then be checked for validity within its deductive framework. For instance, (iii) can be solved by term matching to get \( x = 0, c = y_o, \) and \( d = z_o \) as
the desired substitutions, which is easily shown to be valid by current formula manipulation and simplification systems.

- The loop body code theorem is

\[ \forall y \forall z \forall d \forall c \exists x \exists c' \exists d' [x' + c' \cdot d' = y \cdot z \cap c' < c]. \]

Eliminating quantifiers yields

(i) \[ [c_0 \neq 0 \land x_a = y_0 \cdot z_a] \rightarrow [x' + c' \cdot d' = y_0 \cdot z_0]. \]

The splitting heuristics of PROVER yield the subgoals

(ii) \[ [c_0 \neq 0 \land x_a = y_0 \cdot z_a] \rightarrow x' + c' \cdot d' = y_0 \cdot z_0 \]

and

(iii) \[ [c_0 \neq 0 \land x_a = y_0 \cdot z_a] \rightarrow c' < c_0, \]

which must be proved under the restriction that (ii) and (iii) must be satisfied by the same set of substitutions.

The simplest solution to subgoal (iii) is \( c' = c_0 - 1 \). Trying this in (ii) yields \( c_0 \neq 0 \land x_a + c_0 \cdot d_a = y_0 \cdot z_a \rightarrow x' + c_0 \cdot d_a = y_0 \cdot z_0 \).

Example 2 indicates that heuristics similar to those presented for second-order synthesis in Reference 5 will be helpful to a PROVER-like system. We are currently constructing a PROVER-based synthesis system to use such heuristics.

CONCLUSION

A program \( P \) can be synthesized as a set of programs \( \{P_1, \ldots, P_n\} \) so that the concatenation \( P_1 \cdots P_n \) computes \( F_n \). We have demonstrated that a satisfactory set \( \{P_1, \ldots, P_n\} \) exists such that each member has at most a single loop of the while-do form. Therefore, the synthesis of each \( P_i \), requiring a loop can be stated in terms of proving lemmas of the form of (4) and (5) (or (7) and (8)) for each \( P_i \). Any loop-free \( P_i \) can be directly synthesized from the form of (3) without the need for inductive proofs. If nested loops are desired, the synthesis can be performed hierarchically to expand operations considered primitive during higher-level synthesis.

We have not addressed the problem of mechanically (or otherwise) arriving at a good decomposition and finding loop invariants and variant functions. It has, however, become part of the general lore of programming methodology that it is desirable for a programmer to perform such decompositions before writing programs, and Dijkstra and others have recommended that the loop invariant and variant function be discovered by a programmer before he actually writes a program loop. Thus a successful synthesis system which operates from a specification of the decomposition and associated loop invariants and variant functions is a very desirable goal.

ACKNOWLEDGMENT

The author would like to thank T. W. Pratt and R. T. Yeh for their support and criticism during the early part of this research.

REFERENCES