
MODELS OF INDUCTIVE SYNTHESIS

E. B. KINBER AND A. N. BRAZMA

- ▷ In designing a new algorithm we usually begin with considering a number of examples and then try to generalize them. These examples are mainly input-output examples or sample computations (computation traces, histories of computations) explaining the algorithm's behavior. The aim of our research is to understand and formalize the process of this generalization and eventually to design practical synthesizers. ◁
-

1. THE RECURSIVE-THEORETIC APPROACH

Till the end of 1970s, inductive synthesis was basically studied on the recursive-theoretic level. The most popular and natural recursive-theoretic model of inductive synthesis was introduced by M. Gold in his fundamental paper [15]: given the sequence

$$f(0), f(1), f(2), \dots, f(m), \dots \quad (1)$$

of the values of a recursive function f , it is required to hypothesize an algorithm computing f (later this model was denoted by GN in Soviet work [29], by EX in American work [2, 21], and by LIM in east German work [3]).

The synthesis (inference) procedure in Gold's model is viewed as an infinite process. An inductive inference procedure (called below a strategy), using a sequence (1), produces a sequence of conjectures $M_1, M_2, \dots, M_k, \dots$ such that, for some m_0 , all M_m with $m \geq m_0$ are equal to a correct program (say, a number in a standard enumeration of all partial recursive functions) computing f . M. Gold called this inference process *identification in the limit*. A slightly different model was defined by J. Feldman in [16]: for some m_0 , all M_m with $m \geq m_0$ are correct programs (not necessarily equal) computing f ; this model was denoted by GN^∞ in Soviet papers, and by BC in American ones. Various other models of inductive

Address correspondence to E. B. Kinber, Computing Centre, Latvian University, Riga, Latvia.
Received January 1989.

THE JOURNAL OF LOGIC PROGRAMMING

©Elsevier Science Publishing Co., Inc., 1990
655 Avenue of the Americas, New York, NY 10010

0743-1066/90/\$3.50

inference were defined, investigated, and compared with EX and BC in [17–19].

Identification in the limit has been studied comprehensively from many points of view. We will discuss below some areas of this research.¹

1.1. Complexity of Identification

The number of changes of hypotheses seems to be the most natural measure of complexity for identification in the limit. This measure was studied comprehensively in [20, 22–24]. Let $G^*(\omega, f)$ denote the number of hypothesis changes the strategy G makes in identifying f on an input sequence ω . For any enumerated class of recursive functions (\mathcal{U}, τ) (where τ is a computable enumeration of the class \mathcal{U}) let

$$G^*(\omega, n) = \max_{0 \leq j \leq n} G^*(\omega, \tau_j).$$

Exact estimates for $G^*(\omega, n)$ are obtained in [20, 22]: for any enumerated class (\mathcal{U}, τ) there is a strategy G such that for every input sequence ω

$$G^*(\omega, n) \leq \log_2 n + O(\log_2 n);$$

on the other hand, there is an enumerated class (\mathcal{U}, τ) such that for any identifying strategy G

$$G^*(\omega_0, n) \geq \log_2 n - O(1),$$

where ω_0 is the natural input sequence $f(0), f(1), \dots, f(k), \dots$.

It follows from the above result that there exist enumerated classes of functions having asymptotically the best identifying strategies. On the other hand, it is shown in [24] that any recursive reduction of $G^*(\omega_0, n)$ is possible for some enumerated classes (\mathcal{U}, τ) .

Various possibilities for decreasing the number of hypothesis changes for a given input sequence ω were investigated in [25].

1.2. Identification in the Limit in Nonstandard Enumerations

Classes of functions identifiable in the limit in the standard enumerations may have a complicated structure: for example, they can be not effectively enumerable. In [26, 27] it was investigated how the choice of a nonstandard enumeration influences the identification possibilities. An enumeration of all partial recursive functions was constructed in [27], for which only finite classes of functions were identifiable. Infinite but not effectively enumerable classes of functions are identifiable in a different nonstandard enumeration of all p.r.f.'s [26, 27]. A characteriza-

¹For the most part, work of the research group in Latvian State University is covered.

tion of identifiable classes in terms of Friedberg enumerations and computable one-one enumerations maximal according to certain reducibilities is given in [28].

1.3. Identification in the Limit of Minimal Gödel Numbers

Kinber showed in [29] that minimal Gödel numbers are not inferable in the limit for some recursively enumerable classes of functions. Identification in the limit of minimal numbers was studied comprehensively in [30, 32, 31]. As was shown in [30, 31], possibilities of identifying minimal numbers strongly depend on the choice of a standard enumeration of p.r.f.'s. For example, there is a standard enumeration where just finite classes of functions are identifiable in the limit in the sense of the minimal numbers. Moreover, the set of all standard enumerations is partially ordered with respect to possibilities of identifying the minimal numbers. The number of hypothesis changes for the minimal number's identification also depends on the choice of a standard enumeration [32].

1.4. Probabilistic Identification in the Limit

Probabilistic identifying strategies use a generator of random numbers with a finite alphabet yielding its output values with equal probability and in accordance with a Bernoulli distribution. Classes of recursive functions identifiable with a probability $p > \frac{1}{2}$ have been proved to be identifiable by deterministic strategies [33]. However, probabilistic strategies can be more economical in the sense of the number of hypothesis changes: for every $p < \frac{2}{3}$ and for every $n \in \mathbb{N}$ there is a class identifiable with probability p by a strategy changing hypotheses n times and not identifiable by any deterministic strategy with $2n$ changes of hypotheses [34].

Strategies identifying classes of recursive functions in the limit basically use the method of so-called identification by enumeration: they systematically search the whole space of possible hypotheses until one is found that agrees with all data so far. However, this method is not practical. A possible way to construct more practical inference strategies is to use histories of computations (or sample computations) instead of input-output examples. Some promising results supporting this approach are obtained in [35, 36]. Namely, let ω be a class of automata with a input symbols and b output symbols. Further, let ω be any input sequence of words, $G^*(\omega, A)$ be the number of hypothesis changes necessary for identifying an automaton A on ω , and $G^*(\omega, \mathcal{U}_{a,b}, k) = \max\{G^*(\omega, A) \mid A \text{ has not more than } k \text{ states}\}$. Then there is a strategy G such that for any ω

$$G^*(\omega, \mathcal{U}_{a,b}, k) \lesssim (a - 1)k \log_2 k.$$

A very important corollary follows from this result for the inductive synthesis of programs from their operational-logical histories: there is a strategy G that synthesizes in the limit all programs P from examples of operational-logical histories with at most $\|P\| \log_2 \|P\|$ changes of hypotheses, where $\|P\|$ is the number of all logical commands in P [36]. The number $\|P\| \log_2 \|P\|$ is practically commen-

surable with the number of mistakes a programmer makes designing a program. However, the strategy G still is not practical, because it uses an exhaustive search. A theory and more practical methods of inductive synthesis from sample computations are developed in the next section.

2. SYNTACTICAL INDUCTIVE SYNTHESIS FROM SAMPLE COMPUTATIONS

Synthesis from sample computations explaining the behavior of a program, contrary to identification by enumeration, is based on detection of various regularities in samples. The simplest case is when the input information is regarded as a string of characters without any semantics. Then the synthesis is based on the detection of purely syntactical analogies in examples. The required program is synthesized in a form of a grammar describing all possible computation traces (sample computations). Such a grammar can be considered as a nontraditional way of presenting a program (in fact, a program scheme). To give a semantics to such a program one has to interpret operators and predicates in definite way.

For practical synthesizers it is clear that the class of synthesizable programs must be limited—we have to look for models (schemes) of inductive synthesis applicable to reasonable problems within which the synthesis is effective and which are convenient for the user at the same time. Such a model includes an effective synthesis algorithm. Below we give a survey of the research in inductive syntactical synthesis done by the group at Latvian State University.

2.1. Dot Expressions (J. M. Barzdin [8, 9])

This model was the first model of syntactical inductive synthesis based on detection of fragments of arithmetical progressions in sample computations.

Let us consider the algorithm computing the greatest common divisor (GCD) of two natural numbers X_1 and X_2 . Let $\text{mod}(X_1, X_2)$ be the remainder from integer division of X_1 by X_2 . Then a possible way to describe the GCD algorithm might be the example in Figure 1.

Let us consider another example—the bubble-sort algorithm. It can be described as shown in Figure 2.

From these descriptions (sample computations) evidently it is possible to reconstruct the respective general algorithms. Therefore, a description of the algorithm by sample computations can be regarded as a program in some nontraditional

```

Input  $X_1, X_2$ 
  Let  $X_3 := \text{mod}(X_1, X_2)$ ;
  Is  $X_3 = 0$ ? Suppose it is not.
  Let  $X_4 := \text{mod}(X_2, X_3)$ ;
  Is  $X_4 = 0$ ? Suppose it is not.
  Let  $X_5 := \text{mod}(X_3, X_4)$ ;
  Is  $X_5 = 0$ ? Suppose it is.
Return  $X_4$ ;

```

FIGURE 1

```

Input A: Array (1...4);
  If A(1) ≤ A(2) then; else A(1) ↔ A(2);
  If A(2) ≤ A(3) then; else A(2) ↔ A(3);
  If A(3) ≤ A(4) then; else A(3) ↔ A(4);
  If A(1) ≤ A(2) then; else A(1) ↔ A(2);
  If A(2) ≤ A(3) then; else A(2) ↔ A(3);
  If A(1) ≤ A(2) then; else A(1) ↔ A(2);
Return A;
    
```

FIGURE 2

programming language. How to define the semantics of such a language? A usual way to define the semantics of a language is a translation to another language with semantics known in advance. Since here the intended language is that of examples, it is natural to define its semantics by means of inductive inference rules, reconstructing the general algorithms from their sample computations.

As a means for presenting general algorithms here we will use so-called dot expressions. The basic construction in the language of dot expressions is a *dot term*—any word of the following type:

$$\langle a_1 a_2 \dots a_n \circ \circ \circ b_1 b_2 \dots b_n \rangle,$$

where $a_1 a_2 \dots a_n$ and $b_1 b_2 \dots b_n$ are words in a given alphabet Σ and there exists $c \in N$ such that for all $i \in \{1, \dots, n\}$ one of the following conditions holds: either $a_{ij} b_i \in Z$ and $|a_i - b_i| = c$, or $a_i = b_i$. For example,

$$T_1 = \langle A1 \circ \circ \circ A5 \rangle,$$

$$T_2 = \langle A0(1) \circ \circ \circ A5(6) \rangle$$

are dot terms.

On replacing dots “ $\circ \circ \circ$ ” in a natural way with the appropriate particular string and removing the brackets “ \langle ” and “ \rangle ”, we obtain an *unfoldment* $\text{unf}(T)$ of the given dot term T . For example,

$$\text{unf}(T) = A1A2A3A4A5,$$

$$\text{unf}(T) = A0(1)A1(2)A2(3)A3(4)A4(5)A5(6).$$

Another basic notion is a *dot string*, which we obtain by concatenation of dot terms or embedding one term into another. For example,

$$W_1 = \langle A1 \circ \circ \circ A5 \rangle \langle 1B \circ \circ \circ 6B \rangle C1,$$

$$W_2 = \langle \langle A11 \circ \circ \circ A13 \rangle \circ \circ \circ \langle A31 \circ \circ \circ A33 \rangle \rangle$$

are dot strings.

For dot strings an unfoldment can be defined. To get an unfoldment $\text{Unf}(W)$ of the dot string W we successively replace all its dot terms (starting with the

```

Input  $X_1, X_2$ ;
  <Let  $X_3 := \text{mod}(X_1, X_2)$ ;
    Is  $X_3 = 0$ ? Suppose it is not;  $\dots$ 
    Let  $\underline{XK} := (\underline{XK} - 2, \underline{XK} - 1)$ ;
    Is  $\underline{XK} = 0$ ? Suppose it is not; >
  Let  $\underline{XK} + 1 := \text{mod}(\underline{XK} - 1, \underline{XK})$ ;
  Is  $\underline{XK} - 1 = 0$ ? Suppose it is;
Return  $\underline{XK}$ ;

```

FIGURE 3

outermost) with their unfoldments. For example, $\text{Unf}(W_2)$ is unfolded as follows. First, unfolding the outer term, we get a dot string

$$\langle A11 \dots A13 \rangle \langle A21 \dots A23 \rangle \langle A31 \dots A33 \rangle.$$

Then, after unfolding the rest of the terms, we obtain

$$\text{Unf}(W_2) = A11A12A13A21A22A23A31A32A33.$$

Dot expressions are dot strings depending on a variable (we denote it by K). Thus, for example,

$$E_1 = \langle A1 \dots A\underline{K} \rangle \langle 1B \dots \underline{K} + 1B \rangle C1,$$

$$E_2 = \langle \langle A11 \dots A1\underline{K} \rangle \dots \langle A\underline{K}1 \dots A\underline{K}\underline{K} \rangle \rangle$$

are dot expressions. We define the value $\text{Val}(E, K_0)$ of the dot expression E for the given $K_0 \in N$ as the unfoldment $\text{Unf}(E(K_0))$ of the dot string $E(K_0)$, obtained from the dot expression E by replacing all the substrings $\underline{K} + c$ ($c \in \mathbb{Z}$) with the value $K_0 + c$. Thus, for example, $\text{Val}(E_1, 4) = \text{Unf}(W_1)$ and $\text{Val}(E_2, 3) = \text{Unf}(W_2)$. For an arbitrary $K_0 \in N$, the value $\text{Val}(E, K_0)$ is called a *formal example* of the expression E .

The general algorithm for computing the GCD can be represented as the dot expression in Figure 3. Note, that this dot expression is indeed an algorithm computing the GCD, as it gives all computation traces of the algorithm (for appropriate values of K).

Similarly, the bubble-sort algorithm can be represented as the dot expression in Figure 4.

The essential element of the given model is the system of inductive inference rules reconstructing general expressions from their examples. The system has to be complete in the following sense: if the formal example of the given expression is

```

Input  $A$ : Array  $(1 \dots K)$ 
  <If  $A(1) \leq A(2)$  then; else  $A(1) \leftrightarrow A(2)$   $\dots$ 
    If  $A(\underline{K} - 1) \leq A(\underline{K})$  then; else  $A(\underline{K} - 1) \leftrightarrow A(\underline{K})$   $\dots$ 
  <If  $A(1) \leq A(2)$  then; else  $A(1) \leftrightarrow A(2)$   $\dots$ 
    If  $A(1) \leq A(2)$  then; else  $A(1) \leftrightarrow A(2)$  >>;
Return  $A$ ;

```

FIGURE 4

long enough, then the system of inference rules synthesizes an expression equivalent to the given one. The system consisting of three rules (folding up, standardization, and generalization) described below has been proved to be complete for dot expressions (Brazma [10, 13]). Let us note that the proof is rather complex.

The rule of folding up is based on the notion of the so-called regular strings. A string is said to be regular if it is the unfoldment of some dot term. For example, the strings

$$A1A2A3A4A5 (= \text{unf}(\langle A1 \circ \circ \circ A5 \rangle))$$

and

$$A6B5A5B4A4B3A3B3 (= \text{unf}(\langle A6B5 \circ \circ \circ A3B2 \rangle))$$

are regular. The rule of folding up searches in the given string for the regular substring $\alpha(1)\alpha(2)\dots\alpha(k)$ [or $\alpha(k)\alpha(k-1)\dots\alpha(1)$], $k \in N$, having the maximal value of $k/|\alpha(1)|$ (i.e. the number of repetitions divided by the length of the repeated fragment), and replaces it with the corresponding dot term: $\langle \alpha(1) \circ \circ \circ \alpha(k) \rangle$. The rule of standardization is applied after each application of folding up. It expands maximally and shifts to the right (in a sense) all dot terms of the given dot string. The rule of generalization is applied when no other rule can be applied, and no rule is applied later. It replaces all “large” numbers in a dot string (i.e. numbers which are larger than a half of the maximum) with expressions of the type $\underline{K \pm C}$ ($C \in N$), where K is a variable. Thus, given the string

$$A1A2A3A4A51B2B3B4B5B6BC1,$$

the system synthesizes following dot expression:

$$\langle A1 \circ \circ \circ \underline{AK} \rangle \langle 1B \circ \circ \circ \underline{K+1B} \rangle C1.$$

From the sample computation of the GCD given in Figure 1, the system synthesizes the general algorithm for the GCD given in Figure 3. Similarly, the system synthesizes the bubble-sort algorithm (Figure 4) from the sample computation of the bubble-sort algorithm given in (Figure 2).

From the completeness of the described system of inductive inference rules, it easily follows that there exists an algorithm which synthesizes an arbitrary dot expression in polynomial time.

2.2. WHILE Expressions Constraining Interpreted Predicates

$$(x_1 \leq y_1) \wedge (x_2 \leq y_2) \text{ (Kinber [14])}$$

Loop conditions in the expressions (programs) of this language are interpreted predicates of the type $x \leq y$ and their conjunctions; these predicates do not explicitly occur in sample computations—instead they are interpreted in the computation process. To show the difference from the uninterpreted substrings, we represent them in SMALL CAPITALS:

$$(x := 0) \text{WHILE}(x \leq y) \text{DO}(bx^+)$$

(x^+ denotes the assignment operator $x := x + 1$) is a program in this language, and

$$b0b1b2b3b4b5b6b7$$

is its value (formal sample computation) for $y = 7$.

The standard sort-merge algorithm that merges ordered arrays $A[1:m]$ and $B[1:n]$ into the ordered array $C[1:m+n]$ can be written in this language as follows:

```

R: Input : A : ARRAY(1..x0), B ARRAY(1..y0),
          C : ARRAY(1..x0 + y0),
          (x := 1, y := 1, z := 1)
  WHILE((x ≤ x0) ∧ (y ≤ y0))
  DO(CASE : Is A(x) ≤ B(y)? Suppose yes. Then
      C(z) := A(x), x+z+
      Is A(x) ≤ B(y)? Suppose no. Then
      C(z) := B(y), y+z+
  WHILE(x ≤ x0)DO(C(z) := A(x), x+z+)
  WHILE(y ≤ y0)DO(C(z) := A(y), y+z+)
  Output C.

```

The sample computation

```

Input : A : ARRAY(1..3), B : ARRAY(1..5),
        C : ARRAY(1..8),
[Is A(1) ≤ B(1)? Suppose yes.
  Then C(1) := A(1),
  Is A(2) ≤ B(1)? Suppose yes.
  Then C(2) := A(2),
  Is A(3) ≤ B(1)? Suppose no.
  Then C(3) := B(1),
  Is A(3) ≤ B(2)? Suppose yes.
  Then C(4) := A(3)]
[C(5) := B(2),
  C(6) := B(3),
  C(7) := B(4),
  C(8) := B(5)]
Output C,

```

naturally explaining the algorithm's behavior, is a formal sample of the program R .

Special annotations (brackets [,]) are used in this sample computation to specify approximately the loop boundaries (specification of this kind apparently is not difficult for a user).

A synthesis algorithm has been developed which constructs the necessary WHILE expression from a given system of annotated sample computations describing the program's behavior completely enough. It has been proved that if annotations specify the loop boundaries correctly, then the synthesis algorithm finds a program equivalent to the given one.

The synthesis algorithm handles an arbitrary sample computation in polynomial time. However, the number of samples necessary for the synthesis can be exponential. Nevertheless, if the number of loops in a program is bounded, then the synthesis algorithm has a polynomial time complexity. It would be interesting to know whether it is possible to generalize these results to the case when loop conditions are arbitrary boolean expressions over predicates $x \leq y$.

2.3. FOR *Expressions Containing Interpreted Functions* (Kinber [15])

Programs in this language contain only FOR loops. Additionally, interpreted functions of the type $f(x, y_1, \dots, y_n)$ satisfying certain monotonicity conditions are allowed. For example, using this model one may conveniently formalize the algorithm computing the sum of the first x natural numbers:

T : Input x ; ($z := 1, y := 1$)
 WHILE($z \leq x$)DO($y + z, \text{obtain } y; | z^+$)
 Output y .

The word

Input 5;
 [0 + 1, obtain 1;
 1 + 2, obtain 2;
 3 + 3, obtain 6;
 6 + 4, obtain 10;
 10 + 5, obtain 15;]
 Output 15.

is, in particular, a formal sample computation of T explaining the program's behavior.

A synthesis algorithm has been developed which, given a sufficiently "representative" sample computation, synthesizes an arbitrary program in this language. For instance, from the sample computation given above it synthesizes the program T . The algorithm works in polynomial (in the length of input samples) time.

2.4. *Generalized Regular Expressions* (Brazma and Kinber [5, 6])

The language of generalized regular expressions (g.r.e.'s) is a general model which includes (in a sense) all models considered above (one can obtain these models by putting different restrictions on the g.r.e.'s and fixing classes of interpreted predicates). G.r.e.'s are regular expressions over a set containing an alphabet of variables $X = \{x, y, \dots\}$; an alphabet A ($A \cup X = \emptyset$); expressions $x + c, x \in X, c \in N$; assignment operators $x := y, x := c, x, y \in X, c \in N$; and operators x^+ (to add 1) and x^- (to subtract 1). Examples of g.r.e.'s are

$$P_1 : (x := 0, y := 0)(ax^+x \cup by^+y)^*,$$

$$P : (x := 0)(ax^+x(y := x)(by^+y)^*)^*;$$

To obtain a (formal) sample computation, say, for P_1 , we compute an *unfoldment* of P_1 , for example,

$$\bar{P}_1 : (x := 0, y := 0) ax^+x ax^+x ax^+x by^+y by^+y ax^+x by^+y by^+y by^+y$$

and then get the sample computation

$$a 1 a 2 a 3 b 1 b 2 a 4 b 3 b 4 b 5$$

from \bar{P}_1 by making obvious calculations.

Iteration (*) and union (\cup) correspond respectively to statements of the type
 $\text{WHILE}(P)\text{DO}(C)$

and

$\text{CASE}(P_1 \rightarrow a_1, P_2 \rightarrow a_2, \dots, P_k \rightarrow a_k)$

in real programming languages.

The equivalence and inclusion problems for g.r.e.'s have been proved to be solvable. On the other hand, all recursive functions can be expressed (in a sense) by g.r.e.'s. A class of programs universal (in a sense) for all programs has been defined, and a syntactical synthesis algorithm has been developed for this class. The synthesis algorithm works in polynomial (in the length of input samples) time. The notion of universality is defined in terms of closure with respect to a finite set of "generalizing" transformations over programs.

An interactive synthesis algorithm has been constructed for a wide class of g.r.e.'s. The algorithm can sometimes ask the user (formally, the oracle) whether various expressions are initial fragments of sample computations. The interactive algorithm can also synthesize all universal programs.

2.5. Iterative Programs (Kinber)

A polynomial-time synthesis algorithm has been constructed for a wide class of g.r.e.'s called iterative programs. Iterative programs are those which satisfy the following restriction: one step of any loop's performance increases the value of any variable in the loop at most by 1 (and several other natural restrictions holding for "real" programs). For example, the sort-merge program

$(x := 1, y := 1, z := 1)$ input : a, b ;
 $(a(x) \leq b(y)? \text{ yes, then } a(x) \rightarrow c(z); x^+z^+$
 $\cup a(x) \leq b(y)? \text{ no, then } b(y) \rightarrow c(z); y^+z^+)^*$
 $((a(x) = \Lambda, \text{ then } (b(y) \rightarrow c(z); y^+z^+)^*$
 $b(y) = \Lambda, \text{ then STOP.})$
 $\cup (b(y) = \Lambda, \text{ then } (a(x) \rightarrow c(z); x^+z^+)^*$
 $a(y) = \Lambda, \text{ then STOP.})$ Output : c

is synthesizable in this model, without any annotations, from two "representative" sample computations of the kind

Input : a, b ;
 $a(1) \leq b(1)? \text{ yes, then } a(1) \rightarrow c(1);$
 $a(2) \leq b(1)? \text{ yes, then } a(2) \rightarrow c(2);$
 $a(3) \leq b(1)? \text{ yes, then } a(3) \rightarrow c(3);$
 $a(4) \leq b(1)? \text{ no, then } b(1) \rightarrow c(4);$
 $a(4) \leq b(2)? \text{ no, then } b(2) \rightarrow c(5);$
 $a(4) \leq b(3)? \text{ yes, then } a(4) \rightarrow c(6);$
 $a(5) = \Lambda; \text{ then}$
 $b(3) \rightarrow c(7);$
 $b(4) \rightarrow c(8);$
 $b(5) \rightarrow c(9);$
 $b(6) = \Lambda; \text{ then STOP}$
 Output : c .

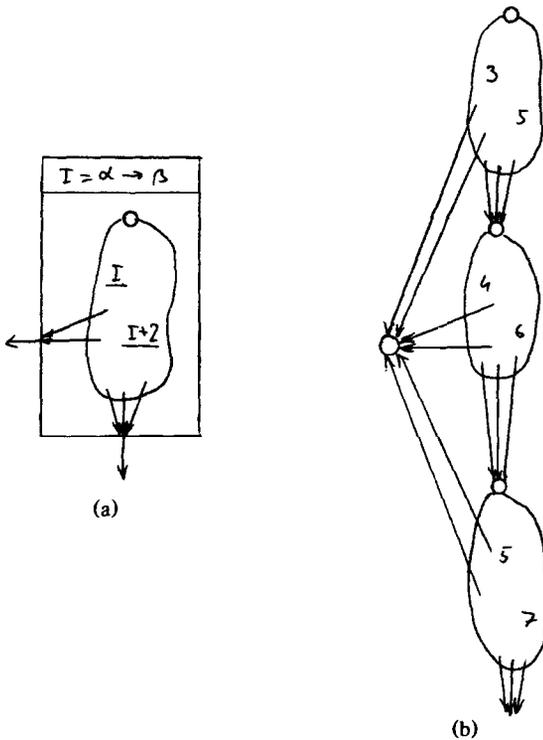


FIGURE 5.

2.6. Graphical Expressions (Etmane [12])

The models of inductive synthesis considered above are linear in the sense that programs and example one-dimensional objects (words) are linear. The graphical expression is a generalization of dot expressions to graphs. For example, the graph given in Figure 5(a) is a graphical expression. Its value for $\alpha = 3$ and $\beta = 5$ is the graph given in Figure 5(b).

The system of inductive inference rules for dot expressions described above can be generalized to graphical expressions, and its completeness can be proved (A. Brazma and I. Etmane). Consequently, there exists an algorithm which synthesizes an arbitrary graphical expression in polynomial time.

For graphical expressions an efficient heuristic method and experimental system of inductive synthesis have been developed by I. Etmane. This system synthesizes, for example, a general algorithm (a graphical expression) solving a linear equation system of n equations for arbitrary n , from sample computations solving a system of four equations.

The sample computations in the considered system of the inductive synthesis are performed directly on arrays monitored on the computer screen by means of light pen. This is important because such sample computation is clear and visual: first we put on the screen input data, for example

$$A = (A(1) = 3, A(2) = 2, A(3) = 4, A(5) = 1);$$

then we put on symbols for the necessary operations (for example, $>$, \leftrightarrow); and

then we create the string " $A(1) \leftrightarrow A(2)$ " by touching with light pen the corresponding expressions $A(1)$, \leftrightarrow , $A(2)$; as a result the values of $A(1)$ and $A(2)$ on the screen are automatically interchanged. This method allows one to present sample computations in many cases more easily and conveniently than the corresponding general algorithms. The idea of performing computations directly on the computer screen was proposed first by A. Biermann [4].

One possibly promising subsidiary direction for the application of inductive synthesis is program optimization. The basic idea is to unfold the loops (i.e. to replace them with particular long linear sample computations), then to optimize the resulting linear programs (their optimization is much easier than that of programs with loops), and finally to reintroduce the loops by folding linear programs using inductive synthesis.

In [8] a different application of inductive synthesis—the synthesis of hypotheses about invariants of loops—is presented.

Finally, it should be noted that translation from all considered models to traditional programming languages is a purely technical problem.

REFERENCES

1. Barzdin, J., Inductive Inference of Automata, Functions and Programs (in Russian), in: *Proceedings of the International Congress of Mathematicians*, Vancouver, 1974, Vol. 2, pp. 455–460.
2. Angluin, D., and Smith, C. H., Inductive Inference: Theory and Methods, *Comput. Surveys* 15(3):237–259 (1983).
3. Klette, R., and Wiehagen, R., Research in the Theory of Inductive Inference by GDR Mathematicians—a Survey, *Inform. Sci.* 22:149–169 (1980).
4. Biermann, A. W. and Krishnaswamy, R., Constructing Programs from Example Computations, *IEEE Trans. Software Engrg.* SE-2:141–153 (1976).
5. Brazma, A. and Kinber, E. B., Generalized Regular Expressions—a Language for Synthesis of Programs with Branching in Loops, *Theoret. Comput. Sci.* 46:175–195 (1986).
6. Brazma, A. and Kinber, E. B., A Language for Non-linear Program Synthesis Containing While-Loops (in Russian), in: *Colloquium "An Intelligence Formalization: Semiotical Aspects"*, Kutaisi, 1985, pp. 173–176.
7. Brazma, A., Inductive Inference of Programs (in Russian), in: *Problems Of Theoretical Cybernetics*, Irkutsk, 1985, pp. 32–33.
8. Barzdin, J. M., Some Rules of Inductive Inference and Their Use for Program Synthesis, in: *Information Processing '83*, North-Holland, 1983, pp. 333–338.
9. Barzdin, J. M., An Approach to the Problem of Inductive Inference (in Russian), in: *Colloquium "Application of Mathematical Logic"*, Tallin, 1983, pp. 156–189.
10. Brazma, A., The Decidability of the Equivalence Problem for Graphical Expressions (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Riga, 1986, pp. 103–155.
11. Barzdin, J. M., On Inductive Synthesis of Programs, in: *Lecture Notes Comput. Sci.*, Vol. 122, 1981, pp. 234–254.
12. Etmane, I., An Experimental System for Presentation of Sample and Synthesis of Programs by Examples (in Russian), No. IPO 017, Latvian State Univ.—Latvian Algorithm and Program Fund.
13. Brazma, A. and Etmane, I., Inductive Synthesis of Graphical Expressions (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Riga, 1986, pp. 156–189.
14. Barzdin, J. M., Brazma, A. N., and Kinber, E. B., Inductive Synthesis of Programs: The State, Problems, Perspectives (in Russian), *Kybernetika*, No. 6, 1987, pp. 81–87.

15. Gold, E. M., Language Identification in the Limit, *Inform. and Control* 10:447–474 (1967).
16. Feldman, J. A., Some Decidability Results in Grammatical Inference, *Inform. and Control* 20:244–262 (1972).
17. Blum, L. and Blum, M., Toward a Mathematical Theory of Inductive Inference, *Inform. and Control* 28:125–155 (1975).
18. Wiehagen, R., Limes-Erkennung Rekursiver Funktionen durch Spezielle Strategien, *EIK* 12:93–99 (1976).
19. Podnieks, K. M., A Comparison of Different Types of Inductive Function Synthesis and Prognostication (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 1, Riga, 1974, pp. 68–81.
20. Barzdin, J. M. and Freivalds, R. V., On Prognostication of General Recursive Functions, *Dokl. Akad. Nauk. SSSR* 206(3):521–524 (1972).
21. Case, J. and Smith, C., Comparison of Identification Criteria for Machine Inductive Inference, *Theoret. Comput. Sci.* 25:193–220 (1983).
22. Barzdin, J. M. and Freivald, R. V., Prognostication and Synthesis in the Limit of Effectively Enumerable Function Classes (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 1, Riga, 1974, pp. 101–111.
23. Barzdin, J. M., Synthesis in the Limit of τ -Indices (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 1, Riga, 1974, pp. 112–116.
24. Barzdin, J. M., Kinber, E. B., and Podnieks, K. M., On Speed-up of Function Synthesis and Prognostication (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 1, Riga, 1974, pp. 117–128.
25. Kinber, E. B., On Limiting Recursive Function Identification Speed-up by Changing Sequence of Questions (in Russian), *EIK* 13(7/8):369–383 (1977).
26. Freivald, R. V., On Limiting Synthesis of General Recursive Function Numbers in Different Computable Enumerations (in Russian), *Dokl. Akad. Nauk SSSR* 219(4):812–814 (1974).
27. Freivald, R. V., Possibilities of General Limiting Recursive Function Synthesis in Different Computable Numberings (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 2, Riga, 1975, pp. 3–25.
28. Freivald, R. V., Kinber, E. B. and Wiehagen, R., Inductive Inference and Computable One-One Numbering, *Z. Math. Logik Grundlag. Math.* 28:463–479.
29. Kinber, E. B., On Limiting Synthesis of Almost Minimal Gödel Numbers, in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 1, Riga, 1974, pp. 221–223.
30. Freivald, R. V., Minimal Gödel Numbers and Their Identification in the Limit, in: *Lecture Notes in Comput. Sci.*, Vol. 32, pp. 219–225.
31. Freivald, R. V. and Kinber, E. B., Limiting Identification of Minimal Gödel Numbers (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 3, Riga, 1977, pp. 3–34.
32. Kinber, E. B., On Limiting Identification of Minimal Numbers for Functions in Effectively Enumerable Classes (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 3, Riga, 1977, pp. 35–56.
33. Freivald, R. V., On the Principal Capabilities of Probabilistic Algorithms in Inductive Inference (in Russian), *Semiotika i Inform.* 12:137–140 (1979).
34. Wiehagen, R., Freivald, R. V., and Kinber, E. B., On the Power of Probabilistic Strategies in Inductive Inference, *Theoret. Comput. Sci.* 28:111–133 (1984).
35. Barzdin, J. M., Prognostication and Limiting Synthesis of Finite Automata (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 1, Riga, 1974, pp. 129–144.
36. Barzdin, J. M., A Note on the Synthesis of Programs from Histories of Their Behavior (in Russian), in: *Colloquium "Theory of Algorithms and Programs"*, Vol. 1, Riga, 1974, pp. 145–151.