USING ATTRIBUTE GRAMMARS FOR DESCRIPTION OF INDUCTIVE INFERENCE SEARCH SPACE

Uğis Sarkans*, Jānis Bārzdiņš**

Institute of Mathematics and Computer Science,
University of Latvia,
Raiņa bulvāris 29, LV-1459, Riga, Latvia

Abstract. The problem of practically feasible inductive inference of functions or other objects that can be described by means of an attribute grammar is studied in this paper. In our approach based on attribute grammars various kinds of knowledge about the object to be found can be encoded, ranging from usual input/output examples to assumptions about unknown object's syntactic structure to some dynamic object's properties. We present theoretical results as well as describe the architecture of a practical inductive synthesis system based on theoretical findings.

1 Introduction

The problem of discovering new proofs, formulas, algorithms etc. usually is solved by some kind of exhaustive search. One of the main issues here is how to minimize the extent of search by using our hypothetical knowledge about the object to be discovered. In this article we will concentrate our attention to synthesis of syntactic objects using various kinds of knowledge about them. If the objects we are trying to synthesize are, e. g., expressions in some fixed signature, then in the simplest case that knowledge will be (after assigning some interpretation to the signature) function values computed on some sample argument values, i.e., usual input/output examples. However, we want to be able to describe also some other properties of the unknown expression (function), i.e., treat the unknown function not as a black box function but as a "gray box" function. These other properties could be either some entirely syntactical properties of the expression we are looking for, or, taking into account also some interpretation, dynamical properties of the function evaluation process.

The question we are seeking answers to in this paper is, how we should present our knowledge so that it would be possible to rapidly examine those and only those objects that match our knowledge? Roughly speaking, the aim of this article is to show that in some sense it is possible to perform such search efficiently enough.

* `usarkans@cclu.lv`
** `jbarzdin@cclu.lv`

Our central aim is synthesis of syntactic objects, i.e., expressions over some fixed signature or programs in some fixed programming language, that can be supplemented by semantic interpretation.

It was understood already long ago that it is convenient to describe such syntactic objects by means of context-free grammars. Then a grammar generates a language with strings belonging to this language being our syntactic objects.

The well-known notion of attribute grammars is linked with the notion of context-free grammars. Our central observation that our approach is based on is that various kinds of hypothetical knowledge about the unknown object usually can be described by means of an attribute grammar that is based on the context-free grammar defining the description space.

As an example, suppose that our description space is defined by a context-free grammar describing the language of simple arithmetic expressions. Then, by supplementing each nonterminal with just a single attribute, we can make an attribute grammar that, for example, counts the number of multiplication operations in expressions, or that can be used to calculate the value of the expression when variable values are fixed, or that limits the values of intermediate expressions. See [3] for more details.

The main problem we will solve is the following: if some attribute grammar is given, is it possible to efficiently enumerate the corresponding language without considering strings that do not belong to it? The aim of this article is to show that, if some conditions hold, it is possible.

The results discussed in this article generalize the results presented in [3], which in turn was generalization of results presented in [1] and [2]. In [3] we considered only attribute grammars with synthesized attributes, and these attributes had to be independent, i.e., equations for computing one attribute could involve only the same attribute of the production right-hand side symbols. In this article we show that a more general class of attribute grammars can be considered, having possibly also inherited attributes and dependencies between different attributes. In some sense the presented article can be regarded as the concluding article in the series that started with [1].

Other approaches to synthesis of expressions include discovery systems BACON ([7], [8]), genetic programming ([4], [5], [6]).

2 Definitions and the Main Result

We suppose that an attribute grammar associates constant values with terminal symbols. For nonterminals attribute values are evaluated by means of corresponding functions. Every production of the grammar has one function for each synthesized attribute of the left-hand side nonterminal and one function for each inherited attribute of each of the right-hand side nonterminal and terminal symbols. If there is a production with several instances of the same nonterminal on the right-hand side, we would write the expression defining the function for computing attributes like this:

$$S \leftarrow AA \ \ / \ \ a_S \leftarrow a_{A_1} - a_{A_2}$$

In article [3] we used conditions — binary predicates that could be attached to every production —, and only inference trees with every predicate being true for every tree node were acceptable. Here predicates will be modeled by partially defined functions, and only inference trees with all attributes defined will be acceptable; see below for more detailed explanation.

By language that is defined by such a grammar we will understand the set of all strings that can be inferred by means of acceptable inference trees.

There is some domain associated with every attribute. From now on we will consider only attributes with finite domains D (for the sake of simplicity we will assume that these domains are subsets of $\mathbb{N}$). That means that arguments of functions for computing attributes, as well as their results belong to D. As we already mentioned, these functions can be partially defined as well. We will say that a grammar is finite if its attributes are of finite domain.

In the following discussion, in order to avoid talking about complexity of attribute evaluation functions, we will assume that each function value, given function arguments, can be computed in constant time.

We will say that an algorithm enumerates a language in setup time T and ith step time T_i, if this algorithm outputs the first string w_1 in time $T + T_1$, and the ith string w_i $(i = 2, 3, \ldots)$ in time T_i from the moment when outputting the previous string w_{i-1} was finished. In this paper by *algorithm* we mean a RAM-machine.

Now let us repeat some definitions in a more formal manner that would be convenient for presenting our results.

Let $G = \langle T, N, P, S\rangle$ is a context-free grammar, where:

- T — finite terminal symbol set;
- N — finite nonterminal symbol set;
- $S \in N$ — start symbol;
- P — a finite production set, where each production $(1 \le k \le |P|)$ is in form

$$A_k \leftarrow B_{k,1}B_{k,2}\ldots B_{k,s(k)}, \text{ where } A_k \in N \text{ and } B_{k,i} \in N \cup T$$

By trees we will denote structures of the form

- $\langle T_i\rangle$, where $T_i \in T$, or
- $\langle N_i, K_1, K_2, \ldots, K_n\rangle$, where $N_i \in N$ and K_j are trees.

We will say that a tree K corresponds to a terminal symbol T_i, if $K = \langle T_i\rangle$. We will say that a tree K corresponds to a production $N_i \leftarrow B_{k,1}B_{k,2}\ldots B_{k,s(k)}$, if $K = \langle N_i, K_{k,1}, K_{k,2}\ldots K_{k,s(k)}\rangle$, where trees $K_{k,i}$ correspond to symbols $B_{k,i}$. We will say that a tree K corresponds to a nonterminal N_i, if it corresponds to some of productions $N_i \leftarrow B_{k,1}B_{k,2}\ldots B_{k,s(k)}$.

Now we will define the terminal string $w(K)$ corresponding to a tree K:

$$w(\langle T_i\rangle) = T_i$$
$$w(\langle N_i, K_1, K_2, \ldots, K_n\rangle) = w(K_1)w(K_2)\ldots w(K_n)$$

A string $w \in T^\star$ can be inferred in grammar G, if there exists a finite tree K_w corresponding to start symbol S such that $w = w(K_w)$; such tree will be called an inference tree of w. The set of all strings that can be inferred in G will be called language $L(G)$. The grammar G is unambiguous, if for every string $w \in L(G)$ there exists only one inference tree K_w. By depth of a string $w \in L(G)$ we will denote the depth of the corresponding inference tree.

Each of the elements $C_k \in N \cup T$ may have several attributes $c_{k,i}$, domains of all attributes are finite subsets of natural numbers. Values of attributes assigned to T elements (terminals) are constants, while values of N (nonterminals) are computed, using attribute evaluation functions. We will denote by $\bar{c}_k$ the tuple of all attributes of C_k, i.e., $(c_{k,0}, \dots, c_{k,j})$.

There will be two kinds of attributes: synthesized and inherited attributes. We will denote synthesized attributes by c^{syn} and inherited attributes by c^{inh}.

Every production has several attribute evaluation functions assigned to it: if the production is in form $C_{i_0} \leftarrow C_{i_1} C_{i_2} \dots C_{i_j}$, then there is a corresponding function for each synthesized attribute of C_{i_0} and for each inherited attribute of $C_{i_1} C_{i_2} \dots C_{i_j}$. For a synthesized attribute $c^{syn}_{i_0,k}$ the evaluation function is $f(\bar{c}_{i_0}, \dots, \bar{c}_{i_j})$, for an inherited attribute $c^{inh}_{i_m,k}$ the evaluation function is $f(\bar{c}_{i_0}, \bar{c}_{i_m})$, i.e., synthesized attributes can depend on all other attributes in the production, while inherited attributes can depend only on other attributes of the same symbol as well as on attributes of the left hand side symbol of the production.

Although there could be the same function used as the attribute evaluation function for several attributes (e.g., the identity function), we will assume that each attribute has its own, separate evaluation function (possibly partially defined). If we regard some evaluation function as defined by means of a table where there is a separate row for each possible argument tuple together with the corresponding function value, then by function volume we will understand the number of rows in this table. By production volume we will understand the sum of volumes of all attribute evaluation functions attached to this production. By volume of the grammar we will call the sum of all grammar production volumes.

A context-free grammar G that is supplemented with attributes and attribute evaluation functions will be called attribute grammar and denoted by G^+.

A tree $\langle T_i \rangle$ has the same attributes as the terminal symbol T_i, and attribute values are the same constants. A tree $\langle C_{i_0}, K_{i_1}, K_{i_2} \dots K_{i_j} \rangle$ that corresponds to the production $P_i = (C_{i_0} \leftarrow C_{i_1} C_{i_2} \dots C_{i_j})$ has the same synthesized attributes as nonterminal C_{i_0}, and attributes are evaluated by first evaluating the values of attributes $\bar{c}_{i_1}, \dots, \bar{c}_{i_j}$ that are assigned to $K_{i_1}, \dots, K_{i_j}$ and then by using the corresponding functions attached to production P_i. Similarly, subtrees K_j have the same inherited attributes as nonterminals C_j.

We will say that a string w can be inferred in grammar G^+, if

- it can be inferred in G, and
- it is possible to compute the values of all attributes assigned to the inference tree and each of its subtrees (it is not always possible, because attribute evaluation functions are partially defined).

By language $L(G^+)$ generated by an attribute grammar G^+ we will understand the set of all strings that can be inferred in G^+.

Theorem 1. *There exists an algorithm which, having received an arbitrary finite unambiguous noncircular attribute grammar, enumerates without repeating the corresponding language in setup time $O(|G^+|^k)$, where $|G^+|$ is the volume of the grammar and k is the maximal number of attributes belonging in the grammar to a single symbol, and ith step time $O(|w_i|)$, where w_i is the string output in the ith step.*

3 Sketch of Proof

We will define the grammar graph $\mathcal{G}_{G^+}$ corresponding to the grammar G^+. It will contain two kinds of nodes: terminal and nonterminal symbol nodes and production nodes.

There will be a symbol node in $\mathcal{G}_{G^+}$ corresponding to each triple $\langle C, a, v\rangle$, where $C \in N \cup T$, a is some attribute belonging to C and v is some value of this attribute. Production nodes will correspond to table rows (equalities) that define G^+ attribute evaluation functions: if the function f that is defined for computing attribute c of production $C_{i_0} \leftarrow C_{i_1} C_{i_2} \ldots C_{i_j}$ is defined by n rows of the form $f(\bar{c}_{i_0}, \bar{c}_{i_1}, \ldots, \bar{c}_{i_j}) = c$, each of these rows will have a corresponding production node p. For each p there will be the following arcs in $\mathcal{G}_{G^+}$ as well. There will be an arc from p to symbol nodes $\langle C_{i_0}, c_{i_{0,k}}, v\rangle$, where $c_{i_{0,k}}$ is an attribute present in the definition of the attribute evaluation function corresponding to p and v is some value of this attribute that is used in the specific table row for p. These symbol nodes will be called upper nodes of the production node p. There will also be arcs from every node $\langle C_{i_m}, c_{i_{m,k}}, v\rangle$ $(m > 0)$ to p where $c_{i_{m,k}}$ is some attribute present in the definition of the attribute evaluation function corresponding to p and v is some value of this attribute that is used in the specific table row for p; these symbol nodes will be called lower nodes of p.

Now we will define the compressed grammar graph $\mathcal{C}_{G^+}$ that corresponds to the grammar graph $\mathcal{G}_{G^+}$. It will also contain symbol nodes and production nodes. In this graph symbol nodes will contain pointers to symbol nodes in grammar graph $\mathcal{G}_{G^+}$, and similarly production nodes will contain pointers to production nodes in the grammar graph $\mathcal{G}_{G^+}$. For each symbol node in $\mathcal{C}_{G^+}$ there will be as many pointers to symbol nodes in $\mathcal{G}_{G^+}$ as there are attributes attached to the corresponding symbol in G^+, and for each production node in $\mathcal{C}_{G^+}$ the number of pointers to symbol nodes in $\mathcal{G}_{G^+}$ will equal the number of attribute evaluation functions attached to the corresponding production in G^+.

The compressed grammar graph $\mathcal{C}_{G^+}$ will contain the following nodes:

- For each terminal symbol in G^+ there will be a single terminal symbol node in $\mathcal{C}_{G^+}$ with pointers to corresponding terminal nodes in $\mathcal{G}_{G^+}$ (for each attribute c of $C \in T$ there is only one node $\langle C, c, v\rangle$ in $\mathcal{G}_{G^+}$).
- If, for some production P with k attached attribute evaluation functions, there are k production nodes p_i in $\mathcal{G}_{G^+}$ such that for every attribute c of the right hand side of P and for all lower nodes of p_i corresponding to c there is a symbol node s_i in $\mathcal{C}_{G^+}$ with pointers to all these nodes, than there is a

production node in $\mathcal{C}_{G^+}$ with pointers to all nodes p_i, and there are arcs in $\mathcal{C}_{G^+}$ from all s_i to this production node.
- If there is a set of symbol nodes S_i in $\mathcal{G}_{G^+}$ such that there is a production node p in $\mathcal{C}_{G^+}$ containing pointers to all lower nodes of s_i, then there is a symbol node in $\mathcal{C}_{G^+}$ containing pointers to all s_i nodes, and there is an arc from p to this symbol node.

For nodes of $\mathcal{C}_{G^+}$ (not necessarily for each of them) we will define weights:

- the weight of terminal symbol nodes is 0;
- if a production node p has lower nodes $s_1, \ldots, s_k$, the weight of this node equals $\max(w_{s_1}, \ldots, w_{s_k}) + 1$, where w_{s_i} are weights of p lower nodes;
- if $p_1, \ldots, p_k$ are production nodes with a common upper node s whose weights are defined and are equal to $w_{p_1}, \ldots, w_{p_k}$, then the weight of s equals $\min(w_1, \ldots, w_k)$, otherwise the weight of s is not defined (i. e., if there is no such production node).

Graph $\mathcal{C}_{G^+}$ will be augmented by dotted arcs according to the following rule. Assume that s is some symbol node and $p_1, \ldots, p_k$ are production nodes with defined weights $w(p_i)$ such that their upper node is s, and they are ordered so that $w(p_1) \leq w(p_2) \leq \cdots \leq w(p_k)$. Then dotted arcs go from s to p_1, from p_1 to p_2,. . ., from p_{k-1} to p_k.

By C we will denote some symbol of G, and by $\bar{v} = (v_1, \ldots, v_k)$ — some vector of its attributes' values. It is easy to see that the weight of the symbol node in $\mathcal{C}_{G^+}$ that corresponds to C and has pointers to nodes $\langle C, c_j, v_j \rangle$ in $\mathcal{G}_{G^+}$ equals the depth of the most shallow inference tree that corresponds to C whose attributes' value vector is $\bar{v}$.

Grammar graph $\mathcal{G}_{G^+}$ construction. Can be performed in time $O(|G^+|)$, because in $\mathcal{G}_{G^+}$ the number of production nodes equals $|G^+|$ and each production node can be added in constant time.

Graph compression. The compressed graph $\mathcal{C}_{G^+}$ will be constructed in several stages. During the first stage for each grammar symbol two attributes will be compressed, obtaining a partial $\mathcal{C}_{G^+}$ with every node containing no more than two pointers to nodes of $\mathcal{G}_{G^+}$. In the consequtive stages other attributes will be added, until the full $\mathcal{C}_{G^+}$ is obtained. For the sake of simplicity here we will briefly consider only how the first stage is carried out. The algorithm will consist of the initial step and iterative step.

Initial step. Terminal symbol nodes of $\mathcal{C}_{G^+}$ are constructed. Weights 0 are assigned to these nodes.

Iterative step. Consider all pairs of production nodes p_1 and p_2 in $\mathcal{G}_{G^+}$ such that no corresponding node in $\mathcal{C}_{G^+}$ is constructed yet, but for all lower nodes in $\mathcal{G}_{G^+}$ there are matching nodes in $\mathcal{C}_{G^+}$. Then a production node p in $\mathcal{C}_{G^+}$ with pointers to p_1 and p_2 can be constructed and its weight can be computed. If there is a symbol node s in $\mathcal{C}_{G^+}$ with pointers to upper nodes of p_1 and p_2, a dotted arc is added in $\mathcal{C}_{G^+}$ from the last production node on the path formed by dotted arcs leaving s to p. Otherwise such s is constructed, and a dotted arc is added from s to p.

The j-th stage of graph compression can be performed in time $O(|G^+|^{j+1})$, hence we get the time complexity estimation of $O(|G^+|^k)$ where k is the maximum number of attributes attached to a single symbol in G^+.

In the terms of Theorem 1 grammar graph construction and graph compression comprise the setup stage, therefore we have shown that time complexity of the setup stage is $O(|G^+|^k)$.

String output. The inference tree K of a string $w \in L(G^+)$ will be called annotated if there is a node of graph $\mathcal{C}_{G+}$ associated with every K subtree K_i according to the rule that, if K_i is in form $\langle S\rangle$, where $S \in T$, or $\langle S, K_{i,1}, \ldots, K_{i,n}\rangle$, where $S \in N$, and the value of its attribute vector is $\bar{v}$, the associated graph node is $\langle S, \bar{v}\rangle$.

We will define the minimal subtree of a symbol node s in $\mathcal{C}_{G+}$ that corresponds to symbol S. If $S \in T$, then the minimal subtree is $\langle S\rangle$. If $S \in N$ and there is no dotted arc leaving s, the minimal subtree is not defined for this node. If $S \in N$ and there is a dotted arc leaving s, then we have to consider the production node p that this arc enters. If this production node corresponds to the production $P \leftarrow S_1 \ldots S_n$, then the minimal subtree of s is $\langle S, K_1, \ldots, K_n\rangle$, where K_i are minimal subtrees of p lower nodes. If for some node the minimal subtree is defined, it is a finite object with depth equal to the weight of this node.

Similarly we define the minimal subtree of a production node of $\mathcal{C}_{G+}$. It will be defined only for production nodes with defined weights. The minimal subtree of a production node will be equal to the minimal subtree of its upper node. The depth of minimal subtrees of production nodes are equal to their weights as well. The number of steps necessary for outputting the terminal string $w(K)$ that corresponds to the minimal subtree K of some node is $O(|w(K)|)$, where $|w(K)|$ is the length of this string.

We will say that for an annotated inference tree K there exists an alternative inference tree, if there is a dotted arc leaving the production node p corresponding to K and entering some production node p'. Then the minimal subtree of node p' will be called alternative inference tree for tree K.

The language $L(G^+)$ that the presented algorithm enumerates can be infinite, therefore enumeration will be performed in a breadth-first manner. A potentially infinite queue will be used for storing marked inference trees. By marked inference trees we will understand annotated inference trees with (possibly) marked subtrees for which there exist alternative inference subtrees.

Assuming that $\mathcal{C}_{G+}$ contains only one node s corresponding to the start symbol S of grammar G^+, the algorithm for enumerating will be as follows.

Initial step.

- The terminal string $w(K)$ that corresponds to the minimal subtree of the node s is output.
- K is entered into the queue, marking all subtrees for which alternative subtrees exist.

Iterative step. While the queue is not empty:

- Take the first inference tree K from the queue.
- If at least two subtrees of K are marked, enter K at the end of the queue, removing the first marker from the left-hand side.
- Replace the first marked subtree of K by its alternative inference subtree, obtaining inference tree K'.
- Output $w(K')$.
- If in K' there is a subtree to the right of the changed point with an alternative, put K' at the end of the queue, marking all alternative points from the changed point (including) to the right.

In the general case when graph $\mathcal{C}_{G^+}$ contains n nodes s_j corresponding to the grammar start symbol S, n separate queues are set up. The words for which the value of their inference tree corresponds to s_1 are output on the first, $n+1$-st, $2n+1$-st etc. steps of the algorithm, the words for which that value corresponds to s_2 are output on the second, $n+2$-nd, $2n+2$-nd etc. steps, etc.

4 Notes on implementation details

The described algorithms are being implemented in a practical inductive inference system. Here we will shortly describe its architecture. For purposes of practical implementation some deviations were made compared to the theoretically "clean" algorithms.

There is a separate module of grammar graph construction and a separate module of graph compression in the system. According to the theoretical algorithms these two modules have to work sequentially in the order that they were just mentioned in. However, we have noticed that for some search spaces it is more efficient to start compressing the graph before it is fully constructed. To be able to organize synthesis process in such manner we implemented a control module which acts as a dispatcher between graph constructor and graph compressor, that can be easily customized for different dispatching strategies.

For real world examples domain node and production node sets become quite large, and we have experimented with several strategies of compressing these sets. In the compression process several domain nodes are merged into a single node. When such compression takes place, the graph loses precision in the sense that it encodes some strings that do not belong to the language. However, if we add to the grammar some more information, e.g., additional input/output examples, incorrect paths through the graph are filtered out. In the case of input/output examples graph nodes can be compressed more safely if they have values further from zero. For experimentation purposes we have implemented a separate domain node writer module that can be easily changed to support different node compression strategies.

A separate module implements graph cleaning procedure. According to our theoretical algorithms graphs are constructed and compressed in a bottom-up manner, and language strings are output in a top-down manner. Compressed graphs can be made smaller by traversing noncompressed graphs in a top-down manner and registering which nodes are reachable; then during compression only reachable nodes are considered. We call this process graph cleaning.

At present inductive inference system implementation is in progress.

5 Conclusion

There were successful computer experiments carried out that employed methods developed in [1] and [2]. In these experiments algebraic expressions were synthesized from input/output examples. In the most successful experiments the formula for the volume of a frustum of a square pyramid and the formula for finding roots of a quadratic equation were synthesized in reasonable time. Algorithms and the synthesis system described in this paper are general and flexible enough to permit easier setup of computer experiments. We believe that computer experiments can bring new insights into inductive synthesis process in our framework and help in obtaining new theoretical results.

References

1. J.M.Barzdin and G.J.Barzdin. Rapid construction of algebraic axioms from samples. Theoretical Computer Science 90. – 1991. – pp. 199–208.
2. J.Barzdins, G.Barzdins, K.Apsitis, U.Sarkans. Towards Efficient Inductive Synthesis of Expressions from input/output Examples. Lecture Notes in Artificial Intelligence, vol.744. – 1993. – pp. 59–72.
3. J.Barzdins, U.Sarkans. Incorporating Hypothetical Knowledge into the Process of Inductive Synthesis. Lecture Notes in Artificial Intelligence, vol.1160. – 1996. – pp. 156–168.
4. J.H.Holland. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. MIT Press, 1992.
5. J.R.Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.
6. J.R.Koza. Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, 1994.
7. P.Langley, G.Bradshaw, H.A.Simon. Rediscovering chemistry with the BACON system. In Machine Learning: An Artificial Intelligence Approach, R.S.Michalski, J.G.Carbonell, T.M.Mitchell (eds.), Tioga Press, Palo Alto, CA, 1983.
8. P.Langley, H.A.Simon, G.Bradshaw. Heuristics for Empirical Discovery. In Computational Models of Learning, L.Bolc (ed.), Springer-Verlag, 1987.

This article was processed using the LaTeX macro package with LLNCS style