INCORPORATING HYPOTHETICAL KNOWLEDGE INTO THE PROCESS OF INDUCTIVE SYNTHESIS

Jānis Bārzdiņš*, Uğis Sarkans**

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

Abstract. The problem of inductive inference of functions from hypothetical knowledge is investigated in this paper. This type of inductive inference could be regarded as a generelization of synthesis from examples that can be directed not only by input/output examples but also by knowledge of, e. g., functional description's syntactic structure or assumptions about the process of function evaluation. We show that synthesis of this kind is possible by efficiently enumerating the hypothesis space and illustrate it with several examples.

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 general descriptions of functions from various hypothetical knowledge about them, including function values on some sample argument values. In the general case the knowledge could be of different nature, e.g., assumptions that the function can be defined as an expression in some fixed signature and the number of specific operators in this expression is limited.

The question that arises here is, can we rapidly examine those and only those functional descriptions that match our knowledge, with an aim to further test them on some additional examples. Roughly speaking, the aim of this article is to show that in some sense it is possible to perform such search efficiently enough.

Now in more detail about our approach. As it was already mentioned above, our central aim is synthesis of functional descriptions. By functional description we will understand a description of the function in some formal language that makes it possible to compute the values of that function. A typical functional description is a definition in the form of an expression over some fixed signature.

* jbarzdin@cclu.lv
** usarkans@ailab.mii.lu.lv

It is possible to think also about some more general kinds of descriptions, like λ-expressions or some fixed programming language.

Already in the dawn of programming it was understood that it is convenient to describe such descriptions by context-free grammars. An example of a grammar describing arithmetic expressions:

$$\begin{aligned}
&S \leftarrow E \\
&E \leftarrow E + T \\
&E \leftarrow E - T \\
&E \leftarrow T \\
&T \leftarrow T * A \\
&T \leftarrow T/A \\
&T \leftarrow A \\
&A \leftarrow (E) \\
&A \leftarrow x \\
&A \leftarrow y
\end{aligned}$$

In the same way we can describe λ-expressions and functional descriptions in other, more complicated languages.

The well-known notion of attribute grammars with synthesized attributes and conditions 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 function usually can be desribed by means of an attribute grammar with synthesized attributes and conditions that is based on the context-free grammar defining the description space.

Let us explain this idea on the previous example. Assume that the hypothetical knowledge about the function to be found says that the number of '*' operators in the defining expression does not exceed 2. Obviously all such expressions can be described by the following attribute grammar:

(CFG productions)	*(Attributes)*	*(Conditions)*
$S \leftarrow E$	$a_S \leftarrow a_E$	$a_S \leq 2$
$E \leftarrow E + T$	$a_E \leftarrow a_E + a_T$	
$E \leftarrow E - T$	$a_E \leftarrow a_E + a_T$	
$E \leftarrow T$	$a_E \leftarrow a_T$	
$T \leftarrow T * A$	$a_T \leftarrow a_T + a_A + 1$	
$T \leftarrow T/A$	$a_T \leftarrow a_T + a_A$	
$T \leftarrow A$	$a_T \leftarrow a_A$	
$A \leftarrow (E)$	$a_A \leftarrow a_E$	
$A \leftarrow x$	$a_A \leftarrow 0$	
$A \leftarrow y$	$a_A \leftarrow 0$	

(Here and in the following examples by a_X we will denote the attribute of CFG symbol X.)

Now let us examine another case. Suppose we know that the unknown function f is such that $f(3,4) = 5$. It is possible to write down this restriction by means of an attribute grammar as well:

$$
\begin{array}{lll}
S \leftarrow E & a_S \leftarrow a_E & a_S = 5 \\
E \leftarrow E + T & a_E \leftarrow a_E + a_T & \\
E \leftarrow E - T & a_E \leftarrow a_E - a_T & \\
E \leftarrow T & a_E \leftarrow a_T & \\
T \leftarrow T * A & a_T \leftarrow a_T * a_A & \\
T \leftarrow T/A & a_T \leftarrow a_T / a_A & \\
T \leftarrow A & a_T \leftarrow a_A & \\
A \leftarrow (E) & a_A \leftarrow a_E & \\
A \leftarrow x & a_A \leftarrow a_x & \\
A \leftarrow y & a_A \leftarrow a_y & \\
 & a_x \leftarrow 3 & \\
 & a_y \leftarrow 4 &
\end{array}
$$

At the end of the article we will show some other examples.

Thus, the main problem we will solve is the following. Suppose that an attribute grammar with synthesized attributes and conditions is given. Is it possible to efficiently enumerate the corresponding language without considering strings that do not belong to it? In the previous examples, is it possible to enumerate only expressions containing no more than 2 multiplications, or expressions that evaluate to 5 if 3 and 4 are substituted for x and y? The aim of this article is to show that, if some conditions hold, it is possible.

The results discussed in this article generelize the results presented in [6] and [7]. In [6] only input/output examples could serve as hypothetical knowledge, while in [7] the notion of expression's weight was introduced that we could use for expressing limited knowledge of the syntactic structure of the target expression. Other approaches to synthesis of expressions include discovery systems BACON ([1], [2]), genetic programming ([3], [4], [5]).

2 Main Theorems

Let us consider attribute grammars with synthesized attributes and conditions. For the sake of simplicity first we will restrict the number of attributes to 1. A very similar example to the previous one, only more extensive usage of conditions:

$$
\begin{array}{lll}
S \leftarrow E & a_S \leftarrow a_E & a_S = 5 \\
E \leftarrow E + T & a_E \leftarrow a_E + a_T & a_E \leq 10 \\
E \leftarrow E - T & a_E \leftarrow a_E - a_T & a_E \leq 10 \\
E \leftarrow T & a_E \leftarrow a_T & a_E \leq 10 \\
T \leftarrow T * A & a_T \leftarrow a_T * a_A & a_T \leq 10 \\
T \leftarrow T/A & a_T \leftarrow a_T / a_A & a_T \leq 10 \\
T \leftarrow A & a_T \leftarrow a_A & a_T \leq 10 \\
A \leftarrow (E) & a_A \leftarrow a_E & a_A \leq 10 \\
A \leftarrow x & a_A \leftarrow a_x & \\
A \leftarrow y & a_A \leftarrow a_y & \\
 & a_x \leftarrow 3 & \\
 & a_y \leftarrow 4 &
\end{array}
$$

In this example intermediate results of computation can not exceed 10.

The attribute grammar can associate constant values with terminal symbols. For nonterminals attribute values are evaluated by means of corresponding functions. In the previous example constants 3 and 4 are associated with a_x and a_y respectively, while other terminals (for example, $'*'$) do not have any attributes. Every production of the grammar has the corresponding function for evaluating the attribute value of the symbol that is on the left-hand side of the production (hence the name 'synthesized attributes'). If there would be a production with several instances of the same nonterminal on the righ-hand side, we would write the expression defining the function for computing attributes like this:

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

Conditions can be associated with productions, they are predicates with the domain equal to that of the attribute of the production's left-hand side. The start symbol will be the left-hand side symbol of the first production.

Acceptable inference trees in such a grammar are only those with nodes such that the corresponding conditions are satisfied. An acceptable and an unacceptable trees are shown on Figure 1, nodes are labelled with grammar symbols and, in parenthesis, with attribute values. The acceptable tree corresponds to the string $x * x - y$, and the unacceptable tree corresponds to the string $x * y - x - y$ in the grammar above. The tree for $x*y-x-y$ is unacceptable because condition at the T-node denoted by a larger filled circle $(a_T \leq 10)$ does not hold.

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

In general functions that are used for attribute evaluation can be partially defined. For example, instead of introducing conditions of the form $a \leq 10$ in the grammar above we could use partially defined functions, e.g., function $+^{\cdot}$:

$$x +^{\cdot} y = \begin{cases} x + y & \text{if } x + y \leq 10 \\ \text{undefined} & \text{otherwise} \end{cases}$$

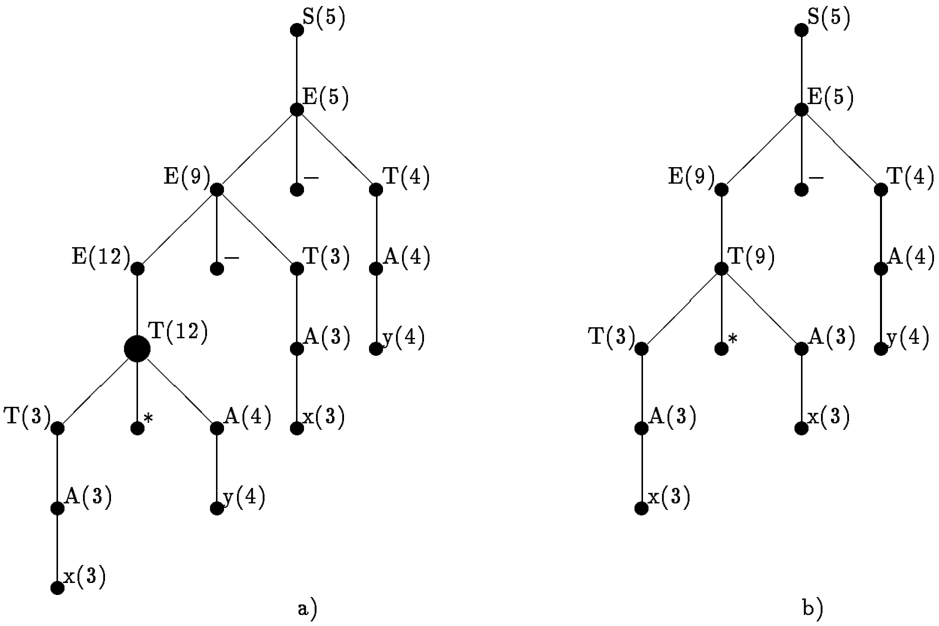

Fig. 1. a) Unacceptable inference tree. b) Acceptable inference tree.

In acceptable inference all conditions are true, and besides that all attribute values can be evaluated (default condition).

There is some domain associated with the attribute. In the previous example it was $\mathbb{N}$, but 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 an attribute grammar is unambiguous, if for every word belonging to the language generated by that grammar there exists only one inference tree. We will say that a grammar is finite if its attributes are of finite domain.

Later we will need the notion of the volume of function $f(x_1, \ldots, x_k)$ for evaluating attributes; this volume will be defined equal to the volume of the set

$$S = \{\langle x_1, \ldots, x_k \rangle | x_1, \ldots, x_k \in D \ \& \ f(x_1, \ldots, x_k) \text{ is defined}\}$$

By volume of the grammar we will understand the cumulative sum of volumes of functions for evaluating attributes of separate productions. Suppose that in our example $D = [1..10]$ and $+, -, *$ and $/$ are the usual partially defined functions with both arguments and result in D. Then it is easy to see that the volume of each of the unary productions (with a single nonterminal on the right-hand side) equals to 10, the volumes of productions with $+$ and $-$ equal to 45, and the volumes of productions with $*$ and $/$ equal to 27. We will define the volumes of productions with only terminals on the right-hand side to be 1; therefore, the volume of the entire grammar is $4*10+2*45+2*27+2*1=186$.

In the subsequent theorems, in order to avoid talking about complexity of attribute evaluation functions and condition predicates, we will assume that algorithms receive as an input function value tables and domain subsets where condition predicates are true. Also, we will assume that the volume of domain D is of the same order of magnitude or less than the volume of the grammar.

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 word w_1 in time $T+T_1$, and the ith word w_i $(i=2,3,\ldots)$ in time T_i from the moment when outputting the previous word w_{i-1} was finished. In this paper by *algorithm* we mean a RAM-machine.

Theorem 1. *There exists an algorithm which, having received an arbitrary finite unambiguous attribute grammar with one synthesized attribute and conditions, enumerates without repeating the corresponding language in setup time $O(|G|)$, where $|G|$ is the volume of the grammar, and ith step time $O(|w_i|)$, where w_i is the word output in the ith step.*

Proof. To make the proof clearer we will shortly re-introduce the notions already discussed from a more formal perspective.

A context-free language $G=\langle T,N,P,S\rangle$:

- 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 notion of a 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 we will call 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 word $w\in L(G)$

there exists only one inference tree K_w. By depth of a word $w \in L(G)$ we will denote the depth of the corresponding inference tree.

Each of the elements $C_k \in N \cup T$ may have an attribute c_k, domains of all attributes are finite subsets of natural numbers. Values of attributes assigned to T elements (terminals) are constants, while values of N attributes are computed, using attribute evaluation functions.

Every production has an attribute evaluation function assigned to it: if the production is in form $C_{i_0} \leftarrow C_{i_1} C_{i_2} \dots C_{i_j}$, then the corresponding function is $f(c_{i_1}, \dots, c_{i_j})$.

Additionally, every production $C_{i_0} \leftarrow C_{i_1} C_{i_2} \dots C_{i_j}$ has a corresponding predicate $R_i : \{c_{i_0} \text{ domain set}\} \to \{True, False\}$ for describing conditions that c_{i_0} must satisfy.

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

A tree $\langle T_i \rangle$ has the same attribute as the terminal symbol T_i, and its value is the same constant. 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 attribute as nonterminal C_{i_0}, and it is evaluated by first evaluating the values of attributes $c_{i_1}, \dots, c_{i_j}$ that are assigned to $K_{i_1}, \dots, K_{i_j}$ and then by using the function f corresponding to production P_i.

We will say that a word w can be inferred in grammar G^+, if it can be inferred in G, and the values of attributes c_i assigned to the inference tree and each of its subtrees are such that the corresponding predicates R_i are true for these values. By language $L(G^+)$ we will understand the set of all words that can be inferred in G^+.

We will define the 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 node in $\mathcal{G}_{G^+}$ corresponding to each pair $\langle C, v \rangle$, where $C \in N \cup T$ and v is some value of the C attribute. Production nodes will correspond to equations that define G^+ attribute evaluation functions: if the function f that corresponds to production $C_{i_0} \leftarrow C_{i_1} C_{i_2} \dots C_{i_j}$ is defined by n equalities of the form $f(c_{i_1}, \dots, c_{i_j}) = c_{i_0}$, each of these equalities will have a corresponding production node p, if $R_i(c_{i_0}) = True$. There will be the following arcs in $\mathcal{G}_{G^+}$ as well. There will be an arc from p to symbol node $\langle C_{i_0}, c_{i_0} \rangle$, this node will be called the upper node of the production node p. There will also be arcs from every node $\langle C_{i_k}, c_{i_k} \rangle$ $(1 \le k \le j)$ that will enter p; these symbol nodes will be called lower nodes of p. For nodes of $\mathcal{G}_{G^+}$ (not necessarily for each of them) we will define weights:

- the weight of terminal symbol nodes is 0;
- if a production node p corresponds to the production $C_{i_0} \leftarrow C_{i_1} C_{i_2} \dots C_{i_j}$, the weight of this node equals $\max(w_1, \dots, w_j) + 1$, where w_i are the weights of p lower nodes;

– if $p_1, \dots, p_k$ are production nodes with a common upper node s whose weights are defined and are equal to $w_1, \dots, w_k$, then the weight of s equals $\min(w_1, \dots, w_k)$, otherwise the weight of s is not defined (i. e., if there is no such production node).

By C we will denote some symbol of G, and by v — some value of its attribute. It is easy to see that the weight of the symbol node that corresponds to pair $\langle C, v\rangle$ equals the depth of the most shallow inference tree that corresponds to C whose attribute value is v and values of all its subtrees' attributes are such that the corresponding conditions hold.

Graph $\mathcal{G}_{G^+}$ will be augmented by dotted arcs according to the following rule. Assume that s is some symbol node and $p_1, \dots, p_k$ 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.

Lemma 2. *Graph $\mathcal{G}_{G^+}$ can be constructed from grammar G^+ in time $O(|G^+|)$.*

Proof. The algorithm consists of the initial step and the iterative step.

Initial step. Weights 0 are assigned to terminal nodes.

Iterative step. Consider all production nodes with weights not yet known, but such that for all their lower nodes weights are known. If there are no such production nodes, the algorithm terminates. Otherwise their weights are computed (in fact, all weights computed during the ith step will be equal to i). If the upper node s of such a production node p does not have a known weight yet, now it can be computed (and it equals to i), and a dotted arc from s to p is added to the graph. Otherwise a dotted arc is added from the production node that is the last on the path formed by dotted arcs and leaving s to p.

It is easy to see that the complexity of this algorithm is $O(|G^+|)$, because the manipulations with every production node can be performed in constant time. □

The inference tree K of a word $w \in L(G^+)$ will be called annotated if there is a node of graph $\mathcal{G}_{G^+}$ 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 $\langle S, K_{i,1}, \dots, K_{i,n}\rangle$, where $S \in N$, and the value of its attribute is v, the associated graph node is $\langle S, v\rangle$.

We will define the minimal subtree of a symbol node s 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 $P \leftarrow S_1 \dots S_n$, then the minimal subtree of s is $\langle S, K_1, \dots, K_n\rangle$, where K_i are minimal subtrees of p lower nodes. It is clear that, if for some node the minimal subtree is defined, it is a finite object with depth equal to the weight of this node — it follows from the construction of $\mathcal{G}_{G^+}$.

Similarly we define the minimal subtree of a production node. 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 present algorithm enumerates can be infinite, therefore enumeration will be performed in breadth-first manner. A potentionally infinite queue will be used for storing marked inference trees. By marked inference trees we will understand annotated inference trees with possibly marked those subtrees for which there exist alternative inference subtrees.

Now it is possible to finish the proof, assuming that the condition predicate R_S that corresponds to the start symbol S of grammar G is true only on a single value v of S attribute. The algorithm for enumerating the language again has the initial step and the iterative step.

Initial step.

- The terminal string $w(K)$ that corresponds to the minimal subtree of node $\langle S, v\rangle$ is output.
- K is entered in 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 R_S is true for several values $v_1, \ldots, v_n$ of S attribute, n instances of the algorithm are executed in parallel with n separate queues: the words for which the value of their inference tree is v_1 are output on the first, $n+1$-st, $2n+1$-st etc. steps of the algorithm, the words for which that value is v_2 are output on the second $n+2$-nd, $2n+2$-nd etc. steps, etc. □

The limitation that there can be only one attribute in the attribute grammar can be removed. If there are several independent attributes $a_1, \ldots, a_k$ with corresponding volumes $|G_1|, \ldots, |G_k|$, the next corollary holds, only then setup time is proportional to the product of all volumes. This corollary follows from the observation that it is possible to encode several attributes into one vector-like attribute.

Corollary 3. *There exists an algorithm which, having received an arbitrary finite unambiguous attribute grammar with k independent synthesized attributes and conditions, enumerates without repeating the corresponding language in setup time $O(|G_1| \times \cdots \times |G_k|)$, where $|G_j|$ is the volume of the corresponding attribute grammar with only jth attribute, and ith step time $O(|w_i|)$, where w_i is the word output in the ith step.*

Obviously the most time-consuming part of this algorithm is the setup step. If there is an attribute that confirms to some additional requirements, the algorithm can be modified, improving the comlexity of the setup step, although at the same time increasing the time necessary for iterative steps.

We will say that a function for computing an attribute is monotonic if it confirms to the following requirements:

1. $f(c_1, \ldots, c_n) \geq \max(c_1, \ldots, c_n)$,
2. If $c'_i \geq c_i$ for every $i \in \{1, \ldots n\}$, then $f(c'_1, \ldots, c'_n) \geq f(c_1, \ldots, c_n)$.

We will call an attribute monotonic if the function for computing it is monotonic. A condition predicate R over domain D will be called monotonic if there exists such $c \in D$ that $\forall x \in D : (x \leq c \;\&\; R(x) = True) \vee (x > c \;\&\; R(x) = False)$. If one of k attributes (see Corollary 3) is monotonic and the corresponding predicates are monotonic as well, the following theorem holds.

Theorem 4. *There exists an algorithm which, having received an arbitrary finite unambiguous attribute grammar with k independent synthesized attributes, one of which (kth) is monotonic, and conditions such that the condition predicates that correspond to the kth attribute are monotonic, enumerates without repeating the corresponding language in setup time $O(|G_1| \times \cdots \times |G_{k-1}| \times \log |G_k|)$, where $|G_j|$ is the volume of the corresponding attribute grammar with only jth attribute, and ith step time $O(|w_i| \times \log |G_k|)$, where w_i is the word output in the ith step.*

The proof of this theorem is based on the proof of the previous theorem and on the ideas of the proof of Theorem 1 in article [7] where only algebraic expressions were considered.

Typical usage of this Theorem would be imposing constraints on the length or depth of the words that we want to be enumerated. In fact, for these and some other simple cases the complexity of the iterative step remains the same as in the Theorem 1, i. e., $O(|w_i|)$ instead of $O(|w_i| \times \log |G_k|)$.

3 Examples

The first example will deal with synthesis of geometric formulas. It is sometimes forgotten that such formulas are typed, with types 'scalar', 'linear', 'area', 'volume' instead of single type 'number'. We can encode type correspondence rules by the following attribute grammar:

$$
\begin{array}{ll}
E \leftarrow (E+E) & a_E \leftarrow \textbf{if } a_{E_1} = a_{E_2} \textbf{ then } a_{E_1} \textbf{ else } \text{“undefined”} \\
E \leftarrow (E-E) & a_E \leftarrow \textbf{if } a_{E_1} = a_{E_2} \textbf{ then } a_{E_1} \textbf{ else } \text{“undefined”} \\
E \leftarrow (E*E) & a_E \leftarrow a_{E_1} + a_{E_2} \\
E \leftarrow (E/E) & a_E \leftarrow a_{E_1} - a_{E_2} \\
E \leftarrow \sqrt{E} & a_E \leftarrow sqrtfun(a_E) \\
E \leftarrow x & a_E \leftarrow 1 \\
E \leftarrow y & a_E \leftarrow 1
\end{array}
$$

$sqrtfun(0) = 0$; $sqrtfun(1) =$ “undefined”
$sqrtfun(2) = 1$; $sqrtfun(3) =$ “undefined”

In this example the domain of attributes is $[0..3]$, and 0 corresponds to scalar expressions, 1 — to linear expressions, 2 — to area expressions and 3 — to volume expressions. In order to make the example easier to understand one more domain value denoted by “undefined” has been added. We will assume that the variables x and y are linear values (lengths), then the grammar describes only expressions with correct type usage, prohibiting, e.g., adding area to volume.

The next example deals with λ-expressions. In order to describe, e. g., some language over $\mathbb{N}^*$ (i. e., consisting of strings of natural numbers) it is very convenient to use λ-expressions describing higher order functions in the definition of the language. For example, the most natural way of describing the language consisting of those and only those strings whose right half elements equal the coresponding squared left half elements (e. g., [1,2,3,4,1,4,9,16]) might look like this:

$$left(w) = map(right(w), \lambda x.x * x)$$

Here the higher order function $map : \mathbb{N}^*, (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}^*$ takes as arguments a string consisting of natural numbers and a function from $\mathbb{N}$ to $\mathbb{N}$ and returns a string over $\mathbb{N}$ obtained by applying the argument function to each element of the argument string. λ is used in such an expression for marking the parameter of the argument function. $left, right : \mathbb{N}^* \rightarrow \mathbb{N}^*$ are functions that return respectively left and right half of the argument string.

The problem is, how to discover such language descriptions from examples (here we are not interested in the type of examples — positive, negative, both positive and negative). By using the results described in this paper it is possible to efficiently enumerate such descriptions that conform to some structural restrictions. After output of each hypothetical description it has to be tested whether language examples match the description.

Language descriptions in the form of equations will be generated by the following CFG:

$$
\begin{aligned}
&A \leftarrow S = S\\
&S \leftarrow w\\
&S \leftarrow left(S)\\
&S \leftarrow right(S)\\
&S \leftarrow map(S, F)\\
&N \leftarrow length(S)\\
&S \leftarrow generate(N)\\
&N \leftarrow (N + N)\\
&N \leftarrow (N - N)\\
&N \leftarrow (N * N)\\
&N \leftarrow (N/N)\\
&N \leftarrow F(N)\\
&N \leftarrow X\\
&F \leftarrow \lambda X.N\\
&X \leftarrow xX\\
&X \leftarrow x
\end{aligned}
$$

Although understanding of semantics of functional symbols is not necessary, let us assume that *length* : $\mathbb{N}^* \rightarrow \mathbb{N}$ returns the length of the argument string, and $generate(n)$ returns $[1, 2, \ldots, n-1, n]$.

Structural restrictions on language descriptions will be imposed by adding attribute evaluation functions and conditions to CFG. The number of variables of type $\mathbb{N}$ in expressions is not pre-set, it is limited only by attribute domain extent. Variables will be of form x, xx, xxx, etc. We want to ensure that equations generated by the grammar do not contain free variables of type $\mathbb{N}$, i. e., for every variable of this type appearing in expression (not under λ) there is an instance of the same variable appearing under λ to the left. For this purpose we can use an attribute with the following evaluation functions and conditions:

$$
\begin{array}{ll}
A \leftarrow S = S & a_{S_1} = a_{S_2} = 0\\
S \leftarrow w & a_S \leftarrow 0\\
S \leftarrow left(S) & a_S \leftarrow a_S\\
S \leftarrow right(S) & a_S \leftarrow a_S\\
S \leftarrow map(S, F) & a_S \leftarrow max(a_S, a_F)\\
N \leftarrow length(S) & a_N \leftarrow a_S\\
S \leftarrow generate(N) & a_S \leftarrow a_N\\
N \leftarrow (N + N) & a_N \leftarrow max(a_{N_1}, a_{N_2})\\
N \leftarrow (N - N) & a_N \leftarrow max(a_{N_1}, a_{N_2})\\
N \leftarrow (N * N) & a_N \leftarrow max(a_{N_1}, a_{N_2})\\
N \leftarrow (N/N) & a_N \leftarrow max(a_{N_1}, a_{N_2})\\
N \leftarrow F(N) & a_N \leftarrow max(a_F, a_N)\\
N \leftarrow X & a_N \leftarrow a_X\\
F \leftarrow \lambda X.N & a_X = a_N \ \& \ a_F \leftarrow a_X - 1\\
X \leftarrow xX & a_X \leftarrow a_X + 1\\
X \leftarrow x & a_X \leftarrow 1
\end{array}
$$

Similarly we can ensure that the statement has at least one occurance of string variable w and that outer simbols of expressions on both sides of equality sign are different (to exclude enumeration of trivial statements like $left(w) = left(w)$)). According to corollary 3 it is possible to efficiently enumerate all statements that conform to all these evaluation function and condition sets.

4 Conclusion

There were succesful computer experiments carried out that employed methods developed in [6] and [7]. 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. We intend to set up similar computer experiments with the approach reported in this paper, making use of not only input/output examples but also of hypothetical knowledge in general. With the algorithms discussed above we should be able to reproduce the results of the previous work and also to synthesize more complex formulas or some other kind of functional descriptions.

References

1. 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.
2. P.Langley, H.A.Simon, G.Bradshaw. Heuristics for Empirical Discovery. In Computational Models of Learning, L.Bolc (ed.), Springer-Verlag, 1987.
3. J.H.Holland. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. MIT Press, 1992.
4. J.R.Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.
5. J.R.Koza. Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, 1994.
6. J.M.Barzdin and G.J.Barzdin. Rapid construction of algebraic axioms from samples. Theoretical Computer Science 90. – 1991. – pp. 199–208.
7. 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.

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