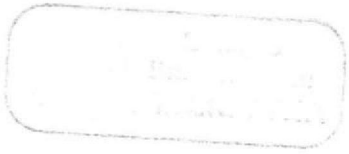
LATVIJAS UNIVERSITĀTE □ UNIVERSITY OF LATVIA

Using Hypothetical Knowledge for Driving Inductive Synthesis

Uģis Sarkans



Thesis for the Dr.Sc.Degree
at University of Latvia

Institute of Mathematics and
Computer Science
University of Latvia
Raiņa blvd. 29
Rīga LV-1459
LATVIA

Rīga, 1998

Acknowledgements

First I would like to thank my supervisor Prof., Dr.hab. Jānis Bārzdiņš for involving me into the exciting field of practical inductive synthesis and continuous support over several years since I was an undergraduate student.

I would also like to thank Prof., Dr.hab. Rūsiņš Freivalds for introducing me to the field of theoretical computer science and teaching me rigorous reasoning about algorithms.

During the first stage of this work I had an opportunity of very productive collaboration with Dr.Guntis Bārzdiņš, Dr.Kalvis Apsītis and Ingars Ribners; I would like to thank them for sharing with me their ideas.

I am grateful to the Scientific Council of Latvia for financial support that allowed me to spend more time on research.

And finally, I would like to hug my family and relatives for bearing with me and allowing me to finish this work.

Contents

1 Introduction

The dissertation is devoted to one of the possible approaches to practically feasible inductive synthesis. In the field of inductive synthesis it is studied how to induce a general description of some object from its particular features. The usual problem is how to synthesize a black-box function from input-output examples.

More often inductive synthesis appears in theoretical context, where the space of possible black-box functions is, e. g., recursive functions and the task is to find the Gödel number of one of them. Our study is more practical in nature, we construct algorithms that can be implemented and tested on real computers. We have been designing efficient search algorithms, avoiding exhaustive search by means of taking full advantage of semantic equality of many considered expressions. This might be the way that people avoid too extensive search when finding proof strategies for theorems, etc.

The formal model for the development of the method has been changing over the years. First we had arithmetic expressions over the domain of natural numbers. Then we proceeded to simple attribute grammars. This generalization was very crucial, as it allowed us to incorporate knowledge about the object to be synthesized in a very general fashion, e. g., to use hypothesis about unknown object's syntactic structure or assumptions about

the process of function evaluation. We have shown that synthesis of this kind is possible by efficiently enumerating the hypothesis space and illustrated it with several examples.

As the last step we introduce a more general class of attribute grammars as suitable for describing inductive synthesis search space. We still have to realize the full potential of the last, general case by making extensive computer experiments.

In this dissertation we follow the development of our method, starting with arithmetic expressions, then considering simple attribute grammars and concluding with more complex attribute grammars. The presented results were published — see [4], [5], [16].

2 Synthesis of arithmetic expressions

2.1 Introduction

Inductive synthesis of recursive functions from input/output examples is a very well studied problem in the recursive-theoretic framework [8] [9]. At the same time few works have been devoted to the problem in the practical perspective because it was considered impossible to synthesize non-trivial functions from input/output examples in the reasonable time.

One of the methods used to synthesize functions from input/output examples is the Occam razor principle stating that we have to search for the

simplest hypothesis, which complies with all available examples. This method could produce very reliable results, but it is difficult to be implemented without exhaustive search.

Nevertheless people are able to guess quite complicated functions from several input/output examples. Many such functions in the form of number sequences are collected in the Angluin's paper "Easily inferred sequences" [1]. The question is: what allows us to generalize such sequences so easily?

In [2] an idea to use algebraic axioms to synthesize functions from input/output examples was suggested. The main advantage of axioms is that they can be synthesized independently of each other and that the complexity of a separate axiom is much smaller than that of the whole program computing the function. If sufficiently many axioms are found, they can describe the function completely. The techniques known in the theory of term rewriting systems can be used to construct an executable program. The computer experiments have shown that algorithms for adding and multiplying binary numbers can be synthesized in this way. The most time-consuming part in such synthesis happens to be the synthesis of expressions in fixed signature which satisfy several input/output examples.

The problem of efficient synthesis of expressions from input/output examples presents interest also by itself. We might wish to be able to induce

some formula, like the one for the volume of the frustum of a square pyramid:

$$V(h,a,b) = h\frac{a^2+ab+b^2}{3}$$

using as input only the results of several measurements, e.g.:

$$V(6,4,2) = 56$$

$$V(3,4,3) = 37$$

$$V(9,2,1) = 21$$

$$V(6,1,3) = 26$$

(This formula is particularly interesting because it was known in the ancient Egypt a long time before Euclid's deductive method in geometry was introduced [11]).

The studies on the problem of efficient inductive synthesis of expressions were initiated in [3] where a reasonably efficient method for such synthesis was proposed. The results of computer experiments illustrating this approach were described in [4]. These experiments have shown that the formulas like the one of the volume of the frustum of the square pyramid can be synthesized on the 33MHz Sparc workstation in about 10 minutes. The ultimate goal was to improve the method by an order of magnitude so that the formulas like the one for solving quadratic equations could be synthesized from input/output examples. In this case the method might become practically interesting.

Here we will consider a new, improved algorithm for inductive synthesis of expressions from input/output examples. In many cases it might be more efficient than the one described in [3]. Then we will briefly describe the first experimental results with this algorithm.

2.2 Definitions and notations

Let the signature Σ be a finite set of functional symbols $\{f_1, \ldots, f_m\}$ where any symbol f_i has a fixed arity. Let $\mathcal{D}$ be a finite domain set. For the sake of simplicity we will assume that $\mathcal{D}$ is a subset of natural numbers. We will say that Σ *is interpreted on* $\mathcal{D}$, if for all functional symbols $f \in \Sigma$ a partially defined function with domain and range in $\mathcal{D}$ is associated with it. By $K_{\Sigma,\mathcal{D}}$ we will denote a particular interpretation of Σ on $\mathcal{D}$. Since we are considering a finite domain $\mathcal{D}$, an interpretation $K_{\Sigma,\mathcal{D}}$ can be completely specified by a finite number of *equalities*. By equality we mean an expression

$$f(a_1, \ldots, a_n) = a_0$$

where $f \in \Sigma$ is a functional symbol of arity n and $a_0, a_1, \ldots, a_n \in \mathcal{D}$.

Example. Let $\Sigma_0 = \{z, s, +\}$ and $\mathcal{D}_0 = \{0, 1, 2, 3\}$. Then a particular interpretation $K_{\Sigma_0,\mathcal{D}_0}$ can be described by the following equalities:

$$\{z = 0, s(0) = 1, s(1) = 2, s(2) = 3,$$

$$+(0,0) = 0, +(0,1) = 1, +(1,0) = 1, +(1,1) = 2, +(2,0) = 2,$$

$$+(0,2) = 2, +(1,2) = 3, +(2,1) = 3, +(3,0) = 3, +(0,3) = 3\}.$$

For other parameter values functions are undefined in this interpretation.

By Σ-*algebra* $\mathcal{K}$ we will understand a set of equalities as in the Example describing an interpretation $K_{\Sigma,\mathcal{D}}$ over a fixed domain $\mathcal{D}$. We assume that $\mathcal{D}$ contains only those domain elements which appear on left-hand or right-hand side of some equality in $\mathcal{K}$. The *volume* of Σ-algebra $\mathcal{K}$ is defined to be the number of equalities in $\mathcal{K}$ and will be denoted $|\mathcal{K}|$. Σ_0-algebra $\mathcal{K}_0$ in the Example above has volume $|\mathcal{K}_0| = 14$.

Let there be a fixed alphabet $\{x_1, x_2, \ldots\}$ of *term variables*. *Open terms* are expressions made of term variables and functional symbols from the signature Σ.

By *weight function* we will understand a mapping from the set of all open terms to the set of all nonnegative integers $w : T(\Sigma) \to N$ with certain properties. Namely, weight function w is defined by means of *auxiliary weight functions* $\{\tilde{f}_1, \ldots, \tilde{f}_m\}$ where $\tilde{f}_j : N^n \to N$ is of the same arity as f_j. We will precisely define open terms and also the weight function:

- Any variable x_i is an open term of weight $w(x_i) = 0$.
- Any 0-arity functional symbol is an open term of constant weight

$$w(f()) = \tilde{f}$$

- If f is a functional symbol of arity n $(n > 0)$ and $t_1, \ldots, t_n$ are open terms, then the expression $f(t_1, \ldots, t_n)$ is an open term of weight

$$w(f(t_1, \ldots, t_n)) = \tilde{f}(w(t_1), \ldots, w(t_n))$$

Auxiliary weight functions $\tilde{f}_j$, $j \in \{1, \ldots, m\}$, must satisfy monotonicity axioms:

A1. $\tilde{f}_j(p_1, \ldots, p_n) > \max(p_1, \ldots, p_n)$.

A2. If $p'_i \geq p_i$ for each $i \in \{1, \ldots, n\}$ then $\tilde{f}_j(p'_1, \ldots, p'_n) \geq \tilde{f}_j(p_1, \ldots, p_n)$.

Besides that functions $\tilde{f}_j$, $j \in \{1, \ldots, m\}$, must be computable from their arguments in constant time.

Some examples of weight function w (over signature $\Sigma_0 = \{z, s, +\}$):

- $\tilde{z} = 1, \tilde{s}(p) = p + 1, \tilde{+}(p_1, p_2) = \max(p_1, p_2) + 1$ (in this case $w(t)$ describes the number of levels in the term),

- $\tilde{z} = 1, \tilde{s}(p) = p + 1, \tilde{+}(p_1, p_2) = p_1 + p_2 + 1$ (in this case $w(t)$ is the number of functional symbols in the term).

Let weight function w is arbitrary fixed, then by *weight* of an open term t we will understand $w(t)$.

The *size* of an open term t is defined to be the number of instances of the functional and variable symbols in t and will be denoted by $|t|$.

The term obtained from an open term t by replacing its variables by elements of domain set $\mathcal{D}$ will be called a *closed term*, and its weight and size are defined to be the same as the weight and size of the corresponding open term t.

We will say that a closed term t can be computed in Σ-algebra $\mathcal{K}$ if its value can be derived by means of elementary equations of $\mathcal{K}$.

Let there be given a tuple $\langle a_1, \ldots, a_k \rangle \in \mathcal{D}^k$, $a_i \neq a_j$ if $i \neq j$[1], and $b \in \mathcal{D}$. Then the pair

$$(\langle a_1, \ldots, a_k \rangle, b)$$

will be called *input/output example*. We will say that an open term t *satisfies* the I/O-example $(\langle a_1, \ldots, a_k \rangle, b)$ in the Σ-algebra $\mathcal{K}$ if:

- t contains no other variables than $x_1, \ldots, x_k$,
- the value of the closed term t' obtained from t by replacing variables $x_1, \ldots, x_k$ by $a_1, \ldots, a_k$ respectively, can be computed in $\mathcal{K}$,
- the value of t' is equal to b.

Let l be a natural number ($l > 0$). We will denote by $A_{\mathcal{K},l}^{(\langle a_1,\ldots,a_k \rangle, b)}$ the set of all open terms such that:

- they satisfy the input/output example $(\langle a_1, \ldots, a_k \rangle, b)$ in $\mathcal{K}$,

[1]This restriction is not essential, it is added only to simplify the explanation of the algorithm.

- they have weight no more than l.

Example. If we consider the Σ_0-algebra $\mathcal{K}_0$ and the weight function $w(t)$ equal to the number of functional symbols in t, then the set $A_{\mathcal{K}_0,2}^{(\langle 1,2\rangle,3)}$ is

$$\{+(x_1,x_2),\ +(x_2,x_1),\ +(x_1,s(x_1)),\ +(s(x_1),x_1),\ +(x_1,+(x_1,x_1)),$$
$$+(+(x_1,x_1),x_1),\ s(x_2),\ s(s(x_1)),\ s(+(x_1,x_1))\}.$$

2.3 Synthesis from one example

We will say that an algorithm having received the input U enumerates the set of objects $\{w_1, w_2, \ldots, w_s\}$ in *setup time* T and ith *step time* T_i, if this algorithm outputs ("prints" on the output tape) the first object w_1 in $T+T_1$ time, and the ith object $w_i (i = 2, 3, \ldots)$ in T_i time from the moment when the previous object w_{i-1} was output. In this paper by *algorithm* we mean a RAM-machine.

Theorem 1 *Let signature Σ and weight function w be fixed. There exists an algorithm which, given any Σ-algebra $\mathcal{K}$, any input/output example $(\langle a_1, \ldots a_k\rangle, b)$ and any natural number $l > 0$, enumerates without repeating the set of terms $A_{\mathcal{K},l}^{(\langle a_1,\ldots a_k\rangle,b)}$ in setup time $O(|\mathcal{K}| \log l)$ and ith-step time $O(|t_i| \log l)$, where t_i is the term output during the ith step $(i = 1, 2, \ldots, |A_{\mathcal{K},l}^{(\langle a_1,\ldots,a_k\rangle,b)}|)$.*

Proof. For Σ-algebra $\mathcal{K}$ we define the corresponding Σ-algebra graph $G_{\mathcal{K}}$. Σ-algebra graph $G_{\mathcal{K}}$ contains nodes of two types: domain nodes denoted by $D_{\mathcal{K}}$ and functional nodes denoted by $F_{\mathcal{K}}$. To distinguish nodes of these two types, in the figures we will show domain nodes as dots and functional nodes as small circles. Domain nodes will correspond to the elements of domain set $\mathcal{D}$ of Σ-algebra $\mathcal{K}$. Functional nodes will correspond to elementary equations of Σ-algebra $\mathcal{K}$ and will be marked by the functional symbol on the left-hand side primitive term in this equation. For any equation

$$f(a_1, \ldots, a_n) = b$$

of Σ-algebra $\mathcal{K}$ the following arcs are added to the graph $G_{\mathcal{K}}$. From the functional node v corresponding to this equation an arc is drawn to the domain node b; the node b is called the *upper node* for the functional node v. From domain nodes $a_1, \ldots, a_n$ arcs (marked by numbers $1, 2, \ldots, n$ respectively) are drawn to the functional node v; domain nodes $a_1, \ldots, a_n$ are called *lower nodes* for the functional node v.

Example. The Σ_0-algebra graph $G_{\mathcal{K}_0}$ which corresponds to the Σ_0-algebra $\mathcal{K}_0$ given above is shown in Fig.1.

Let there be some tuple $\langle a_1, \ldots, a_k \rangle \in \mathcal{D}^k$ such that $a_i \neq a_j$ if $i \neq j$, and let there be some natural number l. In this case we define weights for nodes of the graph $G_{\mathcal{K}}$ according to the following conditions (weight will not be

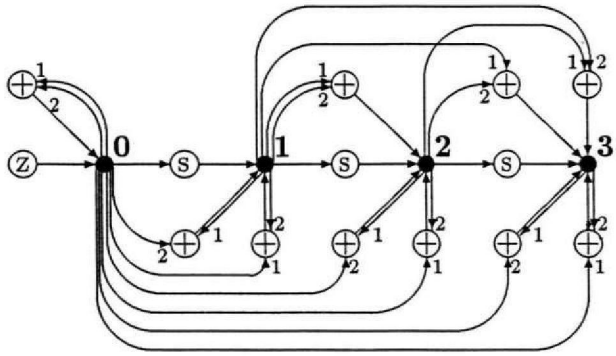


Figure 1: Σ_0-algebra graph $G_{\mathcal{K}_0}$.

defined for all nodes):

- Domain nodes $a_1, \ldots, a_k$ have weight 0.
- Any functional node which corresponds to 0-arity functional symbol f has weight $\tilde{f}$.
- If a functional node v corresponds to some n-arity functional symbol f and $d_1, \ldots, d_n$ are lower nodes of the node v and for all of them weights $h_1, \ldots, h_n$ are defined, then the weight of the node v is $\tilde{f}(h_1, \ldots, h_n)$; otherwise the weight for the node v is not defined.
- Let $v_1, \ldots, v_n$ be functional nodes for which the upper domain node is d and weights $h_1, \ldots, h_n$ are defined. In this case the weight of the domain node d is $\min(h_1, \ldots, h_n)$; otherwise the weight of the domain node d is not defined (i.e. if no such functional nodes v_i exist).

- All weights do not exceed l.

It is easy to see that the weight related this way to some domain node d is exactly the smallest weight among weights of closed terms which in Σ-algebra $\mathcal{K}$ satisfy the pair $(\langle a_1, \ldots, a_k \rangle, d)$, if it does not exceed l. (According to axiom A2, if $t = f(t_1, \ldots, t_k)$ and minimal weights of terms $t_1, \ldots, t_k$ are, respectively, $\tilde{t_1}, \ldots, \tilde{t_k}$, minimal weight of t is

$$\tilde{t} = \tilde{f}(\tilde{t_1}, \ldots, \tilde{t_k}).)$$

Additional arcs, called *dotted arcs* (they are denoted by dotted lines), are added to the graph $G_{\mathcal{K}}$ in the following way. Let d be some domain node for which weight is defined. Let $v_1, \ldots, v_n$ be functional nodes whose upper node is d and for which weights are defined and let these nodes $v_1, \ldots, v_n$ be already ordered according to their weights (weight$(v_i) \leq$weight(v_{i+1})). Then dotted arcs are drawn to connect the node d to v_1, the node v_1 to v_2, ..., the node v_{n-1} to v_n.

The graph $G_{\mathcal{K}}$ which is supplemented by weights and dotted arcs according to some fixed tuple $\langle a_1, \ldots, a_k \rangle$ and fixed l will be called *an annotated Σ-algebra graph* and will be denoted by $G_{\mathcal{K},l}^{\langle a_1, \ldots, a_k \rangle}$.

Example. The graph $G_{\mathcal{K}_0,3}^{\langle 1,2 \rangle}$ which corresponds to the Σ_0-algebra graph $G_{\mathcal{K}_0}$ is shown in Fig.2.

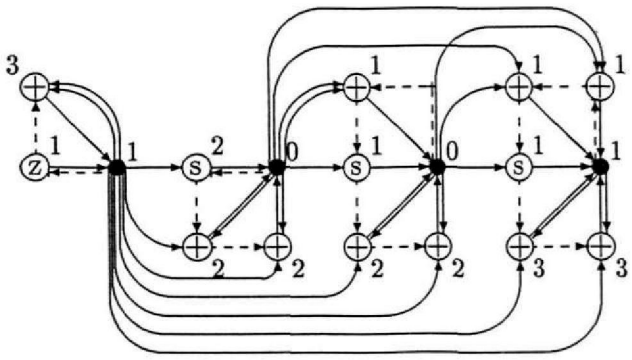


Figure 2: Annotated Σ_0-algebra graph $G^{\langle 1,2\rangle}_{\mathcal{K}_0,3}$.

2.3.1 Construction of annotated sample graph

Lemma 2 *The annotated Σ-algebra graph $G^{\langle a_1,\ldots,a_k\rangle}_{\mathcal{K},l}$ can be built from Σ-algebra $\mathcal{K}$ in time $O(|\mathcal{K}|\log l)$.*

Proof. The algorithm consists of initial step, iterative step and final step.

Initial step. For every domain node $a_i, i = 1,\ldots,k$, we introduce an additional 0-arity functional node with the upper node a_i and set its weight equal to zero. For every "proper" 0-arity functional node f_i we set the weight to $\tilde{f}_i$, if and only if $\tilde{f}_i \leq l$.

Iterative step. First let us examine the graph after the ith iterative step. (We will refer to the initial step as the 0-th step.) There are four kinds of functional nodes.

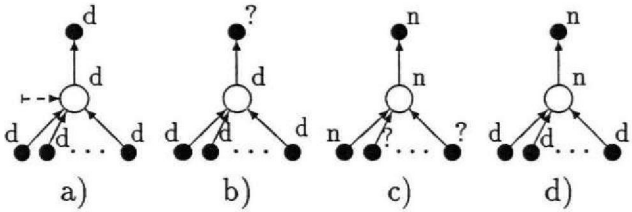


Figure 3: Types of functional nodes after iterative step.

- Node with a dotted arc entering it. Weights of its lower nodes, upper node and itself are defined (Fig. 3a). We denote the set of all such nodes by A_i.

- Node without a dotted arc entering it, but with defined weight. Weights of the lower nodes are defined (Fig. 3b). The set of all such nodes is denoted by B_i.

- Node with undefined weight. The weight of the upper node is also undefined. At least one lower node has undefined weight (Fig. 3c). The set of all such nodes will be called C_i.

- Node with undefined weight, but all lower nodes has defined weight (Fig. 3d). The corresponding set is denoted by D_i.

After the initial step only B_0 and C_0 are not empty.

And now the iterative step itself.

While B_i is not empty, we take from it the functional node with minimal weight (any, if there are several ones) f_i. Two cases should be distinguished.

1. The weight of f_i upper node d_i is defined. Then we add f_i to the end of the dotted path associated with d_i.

$$B_{i+1} = B_i \setminus \{f_i\}, A_{i+1} = A_i \cup \{f_i\}, C_{i+1} = C_i, D_{i+1} = D_i$$

2. The weight of f_i upper node d_i is not defined. We set it equal to the weight of f_i and draw a dotted arc from d_i to f_i.

$$A_{i+1} = A_i \cup \{f_i\}$$

Afterwards we examine all functional nodes with d_i as a lower node (they belong to C_i). Suppose there are k_i functional nodes $g_{i,0}, \ldots, g_{i,k_i}$ for which all lower nodes have defined weights. (For any node $g_{i,j}, j = 0, \ldots, k_i$, d_i was the last node with undefined weight.) Now we can compute $\tilde{g}_{i,0}, \ldots, \tilde{g}_{i,k_i}$. Let us suppose that there is l_i such that for any $j = 0, \ldots, l_i$, $\tilde{g}_{i,j} \leq l$, and for any $j = l_{i+1}, \ldots, k_i$, $\tilde{g}_{i,j} > l$. We set weights for nodes $g_{i,0}, \ldots, g_{i,l_i}$.

$$B_{i+1} = (B_i \setminus f_i) \cup \{g_{i,j} | j = 0, \ldots, l_i\},$$

$$C_{i+1} = C_i \setminus \{g_{i,j} | j = 0, \ldots, k_i\},$$

$$D_{i+1} = D_i \cup \{g_{i,j} | j = l_i + 1, \ldots, k_i\}.$$

It is important to notice that, if every node in A_i has weight not greater than any node in B_i has, the same property holds also for A_{i+1} and B_{i+1}

(because f_i is the node with the minimal weight and for every $j = 0, \ldots, k_i$, $\tilde{g}_{i,j} > \tilde{f}_i$ — axiom A1). Therefore every dotted path is correct, i. e. nodes along the path are ordered according to their weights.

Final step. We remove additional 0-arity functional nodes that were added during the initial step. (It is necessary to delete them correctly from the dotted paths.) These nodes were added during the initial step only to make explanation of the iterative step easier.

We must be able to find a functional node in B_i with minimal weight. We will organize set B_i so that it would be possible to insert, delete and find a functional node with minimal weight in time $O(\log l)$. For this purpose a priority queue [6] will be used.

Let us analyze the time complexity of our algorithm.

Initial step. The number of domain nodes and 0-arity functional nodes is $O(|\mathcal{K}|)$, adding to B_0 takes time $O(\log l)$ for every node. Therefore initial step takes time $O(|\mathcal{K}| \log l)$.

Iterative step. The number of iterative steps is $O(|\mathcal{K}|)$. Although an individual step may require time greater than $O(\log l)$ (if there are many functional nodes with d_i as a lower node), total time required by iterative steps is $O(|\mathcal{K}| \log l)$. (Every node can be moved from C_i to B_{i+1} only once and from B_i to A_{i+1} also only once.)

Final step. It takes time $O(|\mathcal{K}|)$. □

2.3.2 Term output

Let there be a graph $G_{\mathcal{K},l}^{\langle a_1,\ldots,a_k\rangle}$ placed in memory so that its nodes can be accessed by their addresses.

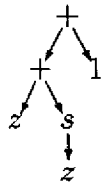
A closed term will be called an *α-term* if:

- it contains no other domain symbols than $a_1, \ldots, a_k$,
- it belongs to the Σ-algebra $\mathcal{K}$.

An α-term will be called *annotated* if, for each of its symbols, the address of some node in the graph $G_{\mathcal{K},l}^{\langle a_1,\ldots,a_k\rangle}$ is attached according to the following rules:

- domain symbols have addresses of the corresponding domain nodes attached to them,
- if $f(g_1, \ldots, g_n)$ is a subterm of the term t and $v_1, \ldots, v_n$ are domain nodes which represent values of closed terms $g_1, \ldots, g_n$, then the symbol f has the address of the functional node marked by f and having lower nodes $v_1, \ldots, v_n$ attached to it.

It is easy to see that any α-term can be annotated.



$+(+(z(), s(z())), 1)$

Figure 4: Tree and linear representation of the same α-term.

We will use two representations of α-terms— ordered tree form and linear form in prefix notation (see Fig. 4). Subterms of the same level which are included in brackets of the same functional symbol (their parent in the tree) will be called siblings. Siblings are ordered according to the number of argument they represent. Therefore we can speak of right siblings for a given subterm. (Such trees and linear layouts for current α-terms will be stored apart from the Σ-algebra graph $G_{\mathcal{K},I}^{(a_1,\ldots,a_k)}$.)

We define the *minimal α-term* of a domain node d in the following way. If $d \in \{a_1, \ldots, a_k\}$, then the symbol d itself is the minimal α-term of the node d. Otherwise we consider the functional node v which is connected to the node d by a dotted arc (if there is no dotted arc from the node d, then the minimal term is not defined). If the node v corresponds to a 0-arity functional symbol f, then this symbol itself is the minimal term. If the node v corresponds to an n-arity ($n > 0$) functional symbol f, then the minimal term is the term $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are minimal α-terms of the lower nodes of the node v.

It is easy to see that the minimal α-term of the node d is a closed term which satisfies the pair $((a_1, \ldots, a_k), d)$ in Σ-algebra $\mathcal{K}$ and has minimal weight which is the weight of the node d in the Σ-algebra graph.

Now we will also define the minimal α-term for functional nodes. Let v be some functional node for which the weight is defined (otherwise the minimal term for v is not defined). Let the functional node v correspond to the functional symbol f of arity n. If $n = 0$, then the symbol f itself is the minimal term of the node v. Otherwise there are lower nodes $d_1, \ldots, d_n$ for the node v and minimal terms $t_1, \ldots, t_n$ for those domain nodes. Then the term $f(t_1, \ldots, t_n)$ is the minimal term of the functional node v. It is easy to see that the weight of the minimal term of the functional node v is equal to the weight of the node v in the Σ-algebra graph.

Lemma 3 *Inequality*

$$\tilde{f}(p_1, \ldots, p_{i-1}, x, p_{i+1}, \ldots, p_n) \leq l$$

can be solved in time $O(\log l)$ *and solution, if it exists, is in form*

$$x \leq l'$$

($p_1, \ldots, p_n, l, l', x$ *are natural numbers,* $l' < l$*).*

Indeed, A2 implies that together with l' all $x < l'$ suit inequality as well. A1 implies that l' (if it exists at all) should be smaller than l. The exact

value of l' can be found in $O(\log l)$ steps of dichotomy.[2] $\square$

For the representation of a given α-term we attach *actual* and *maximal weights* to every ordered tree node (or, equivalently, to every instance of functional or variable symbol) in the following way:

- for the root symbol the maximal weight l is attached to it,
- if a parent node f has its maximal weight l', then its i-th child node has maximal weight l'', where $x \leq l''$ is the solution of inequality

$$\tilde{f}(p_1, \ldots, p_{i-1}, x, p_{i+1}, \ldots, p_n) \leq l'$$

($p_1, \ldots, p_n$ are actual weights of sibling nodes). In the case solution does not exist, we assume $l'' = -1$.

Maximal weight shows how far we can raise the weight of a given subterm without making other changes.

Actual weights are computed upwards in $O(|t|)$ time, and after that maximal weights are computed downwards in time $O(|t| \log l)$.

We will say that there exists an *l'-alternative* for an annotated α-term $t = q(\ldots)$ in the graph $G_{\mathcal{K},l}^{\langle a_1,\ldots,a_k\rangle}$ if from the node q_1 there is a dotted arc to the node q_2 of weight l'. In this case the minimal α-term of the node q_2 is called the *l'-alternative* of term t.

[2]Some important weight functions allow solving the inequality in constant time.

The following *ordering of subterms* for a term t is defined: we say that a subterm t_1 is *left* from the subterm t_2 if the root symbol of t_1 is left from the root symbol of t_2 in the usual linear layout of the term t.

We mark the current α-term in the following way. A special symbol, say "$*$", is associated with the rightmost subterm t_1 of the term t (according to the ordering of subterms defined above) for which an l'-alternative exists, where l' is not greater than the maximal weight assigned to this subterm in the tree representation of t.

It is easy to see that in order to mark an annotated α-term t or to detect it as unmarkable, $O(|t|)$ steps are necessary.

For a marked α-term t we define the *succeeding α-term* t' in the following way. Let t_1 be the subterm marked by "$*$" and t_2 be the alternative of the term t_1. The succeeding term is obtained from the term t by replacing its subterm t_1 by the subterm t_2 and all its right siblings by the minimal terms of the same value.

The next lemma follows easy from the facts already proved.

Lemma 4 *The number of steps necessary to construct from a marked α-term t, its succeeding α-term t' and to mark it is $O(|t'| \log l)$.*

The algorithm mentioned in Theorem 1 works as follows. Having received as input a Σ-algebra $\mathcal{K}$, a pair $(\langle a_1, \ldots, a_k \rangle, b)$ and $l > 0$, first it constructs

the graph $G_{\mathcal{K},l}^{\langle a_1,\ldots,a_k\rangle}$. It takes $O(|\mathcal{K}|\log l)$ steps. After that the algorithm outputs the minimal term t_1 of the node b (in which the different domain values $a_1,\ldots,a_k$ are replaced by variables $x_1,\ldots,x_k$ respectively); this takes $O(|t_1|)$ steps. Then for the term t_2 which is the succeeding term of t_1 the corresponding open term is output (according to the last lemma it takes $O(|t_2|\log l)$ steps), etc. We proceed while marking is possible. It is easy to see that in this way the algorithm enumerates the set $A_{\mathcal{K},l}^{(\langle a_1,\ldots,a_k\rangle,b)}$ without repeating in the time mentioned in Theorem 1.□

2.4 Synthesis from several examples

In the previous section the case where an open term satisfies one input/output example

$$(\langle a_1,\ldots,a_k\rangle,b)$$

was considered. Now we will consider the case where an open term has to satisfy several input/output examples simultaneously. More precisely, let there be a set of input/output examples belonging to Σ-algebra $\mathcal{K}$:

$$Q=\{(\langle a_1^1,\ldots,a_k^1\rangle,b^1),(\langle a_1^2,\ldots,a_k^2\rangle,b^2),\ldots,(\langle a_1^s,\ldots,a_k^s\rangle,b^s)\}.$$

We will say that an open term t *satisfies* a set Q in Σ-algebra $\mathcal{K}$, if t satisfies all elements of Q in $\mathcal{K}$.

Let us denote by $A_{\mathcal{K},l}^{Q}$ the set of all open terms which in Σ-algebra $\mathcal{K}$ satisfy a set Q and have weight no more than l ($l>0$). In other words,

$$A^Q_{\mathcal{K},l} = A^{(\langle a^1_1,\ldots,a^1_k\rangle,b^1)}_{\mathcal{K},l} \cap \ldots \cap A^{(\langle a^s_1,\ldots,a^s_k\rangle,b^s)}_{\mathcal{K},l}.$$

In real situations it is typical that separate sets

$$A^{(\langle a^1_1,\ldots,a^1_k\rangle,b^1)}_{\mathcal{K},l}, A^{(\langle a^2_1,\ldots,a^2_k\rangle,b^2)}_{\mathcal{K},l}, \ldots$$

are of relatively large size but their intersection is of relatively small size. Therefore an important question arises: can elements of $A^Q_{\mathcal{K},l}$ be found directly (i.e. enumerated sufficiently fast) without constructing all s sets?

The next theorem gives a positive answer to this question in some sense.

Theorem 5 *Let signature Σ and weight function w be fixed. There exists an algorithm which, given any Σ-algebra $\mathcal{K}$, any set Q of input/output examples and any natural number $l > 0$, enumerates without repeating the set of terms $A^Q_{\mathcal{K},l}$ in setup time $O(|\mathcal{K}|^{|Q|})$ and ith-step time $O(|t_i|)$ where t_i is the output term in the step i ($i = 1, 2, \ldots, |A^Q_{\mathcal{K},l}|$).*

Proof. For the sake of simplicity we will consider the case when the set Q contains only two examples: $(\langle a^1_1,\ldots,a^1_k\rangle, b^1)$ and $(\langle a^2_1,\ldots,a^2_k\rangle, b^2)$.

For Σ-algebra $\mathcal{K}$ we define a *square Σ-algebra* $\mathcal{K}^2$ which describes an interpretation $K_{\Sigma,\mathcal{D}^2}$:

$$(f_i((a_1,a'_1),\ldots,(a_n,a'_n)) = (b,b')) \in \mathcal{K}^2$$

if and only if

$$(f_i(a_1,\ldots,a_n)=b)\in\mathcal{K} \text{ and } (f_i(a'_1,\ldots,a'_n)=b')\in\mathcal{K}.$$

It is easy to deduce the next important equality:

$$A_{\mathcal{K},I}^{\{(\langle a_1^1,\ldots,a_k^1\rangle,b^1),(\langle a_1^2,\ldots,a_k^2\rangle,b^2)\}} = A_{\mathcal{K},I}^{(\langle a_1^1,\ldots,a_k^1\rangle,b^1)} \cap A_{\mathcal{K},I}^{(\langle a_1^2,\ldots,a_k^2\rangle,b^2)} =$$

$$= A_{\mathcal{K}^2,I}^{(((a_1^1,a_1^2),\ldots,(a_k^1,a_k^2)),(b^1,b^2))}$$

To enumerate the set $A_{\mathcal{K},I}^{\{(\langle a_1^1,\ldots,a_k^1\rangle,b^1),(\langle a_1^2,\ldots,a_k^2\rangle,b^2)\}}$ the algorithm described in Theorem 1 can be applied with respect to Σ-algebra $\mathcal{K}^2$. Because $|\mathcal{K}^2|\leq|\mathcal{K}|^2$ the correctness of Theorem 5, when $|Q|=2$, follows immediately. □

2.5 Computer experiments

The advantage of the given algorithm (if compared with the algorithm described in [3]) is that it allows to effectively enumerate not only terms having limited depth, but it also permits to enumerate terms according to more sophisticated criteria, like the number of functional symbols etc. In the case some function (like square root) is not likely to appear in the expression more than once, the new method allows to restrict its appearances by assigning higher weight to the particular function. Another approach could be used to restrict (to some extent) the number of 2-arity functional nodes in the graph, thus limiting the size of graph that tends to grow exponentially. For this purpose we count only 2-arity functional symbols in the term.

First some implementation details. Setup time $O(|\mathcal{K}|^{|Q|})$ and corresponding volumes of graphs still are too large for practical implementations. Another approach was used — not only the weight of terms was restricted with some w, but also the term level was said not to exceed some l (it is easy to see that the method described above works also for such a pair of restrictions). Besides, the annotated sample graph was constructed dynamically, it contained only reachable domain nodes. Reachability of some node here means that

- there is a term of depth not greater than l such that, substituting its variables with values from the input example, the value of the term corresponds to the node;
- the weight of this term does not exceed w;
- there is a term of depth not greater than l such that, substituting some of its variables with the value corresponding to the node and other variables with values corresponding to reachable nodes, the value of the term equals to the value of the output example;
- the weight of this term also does not exceed w. (Note that there is only one input example consisting of several tuples and only one output example (tuple) as described in the previous section).

Computer experiments showed that the graph corresponding to 3 actual examples usually was smaller than the graph corresponding to 2 examples.

Using described approach the formula for the volume of a frustum of a square pyramid was synthesized. The experiment showed that the new algorithm finds the formula several times faster than the algorithm described in [7].

The next formula we tried to synthesize was the formula for finding the greatest root of quadratic equation of the form $x^2 + bx + c = 0$. Unfortunately we were not able to synthesize the formula from input/output examples using 33MHz Sparc workstation in the reasonable time, assuming $\Sigma = \{+, -, *, /, \sqrt{}, 1, 2, 3, 4\}$ and using weight function which counts instances of functional symbols . Therefore the algorithm of Theorem 5 was further modified taking into account the following observation: the outermost function of the formula is one of the 9 functions appearing in Σ, and having fixed one of them we can reduce the depth of the unknown term by 1 level (we have to consider all 9 possible cases of the outermost function). In fact it is reasonable to fix functions appearing in the two outermost levels, yielding 377 cases but reducing the depth of the unknown term by 2 levels. In this way the formula for greatest root of the quadratic equation

$$x = \frac{\sqrt{b^2 - 4c} - b}{2}$$

was synthesized on the 33MHz Sparc workstation in 20 minutes from the following examples:

$$F(3,-4) = 1$$

$$F(-2,-3) = 3$$

$$F(5,4) = -1$$

$$F(-4,3) = 3$$

$$F(-3,-4) = 4$$

$$F(-3,2) = 2$$

$$F(3,2) = -1$$

Limit on the number of functional symbols — 8, limit on term depth — 6, domain — [-16,25]. Analogical attempt without weight restriction failed.

Then we changed domain for [-24,36], and examples were as follows:

$$F(-1,-6) = 3$$

$$F(4,-5) = 1$$

$$F(5,6) = -2$$

$$F(-4,-5) = 5$$

$$F(6,5) = -1$$

$$F(-5,4) = 4$$

We expected to obtain greater annotated sample graphs, as the domain was greater, but surprisingly the correct formula was synthesized in 10 minutes with much smaller graph volumes. Apparently these examples were "better" if compared to the previous ones. Therefore it seems to be hard to make any theoretical estimations of the behavior of annotated sample graphs on various examples.

3 Simple attribute grammars

3.1 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 section we will concentrate our attention on synthesis of general descriptions of functions from various hypothetical knowledge about them, including function values on some sample argument values, i.e., generalizing the results presented in the previous section. In the general case the knowledge could be of different nature, e.g., assumptions that during function computation intermediate values do not exceed some limits.

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 section 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 here will be 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; we already partly considered this particular case. 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 described 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 this section we will present 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 multiplication operations, or expressions that evaluate to 5 if 3 and 4 are substituted for x and y? The aim of this section is to show that, if some conditions hold, it is possible.

The results discussed in this section generalize the results presented in the previous one. There only input/output examples could serve as hypothetical knowledge, and the notion of expression's weight was introduced that we

could use for expressing limited knowledge about the syntactic structure of the target expression.

Other approaches to synthesis of expressions include discovery systems BACON ([14], [15]), genetic programming ([10], [12], [13]).

3.2 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:

$S \leftarrow E$	$a_S \leftarrow a_E$	$a_S = 5$
$E \leftarrow E + T$	$a_E \leftarrow a_E + a_T$	$a_E \leq 10$
$E \leftarrow E - T$	$a_E \leftarrow a_E - a_T$	$a_E \leq 10$
$E \leftarrow T$	$a_E \leftarrow a_T$	$a_E \leq 10$
$T \leftarrow T * A$	$a_T \leftarrow a_T * a_A$	$a_T \leq 10$
$T \leftarrow T/A$	$a_T \leftarrow a_T/a_A$	$a_T \leq 10$
$T \leftarrow A$	$a_T \leftarrow a_A$	$a_T \leq 10$
$A \leftarrow (E)$	$a_A \leftarrow a_E$	$a_A \leq 10$
$A \leftarrow x$	$a_A \leftarrow a_x$	
$A \leftarrow y$	$a_A \leftarrow a_y$	
	$a_x \leftarrow 3$	
	$a_y \leftarrow 4$	

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 right-hand side, we would write the expression defining the function for computing attributes like this:

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

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 5, nodes are labeled 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.

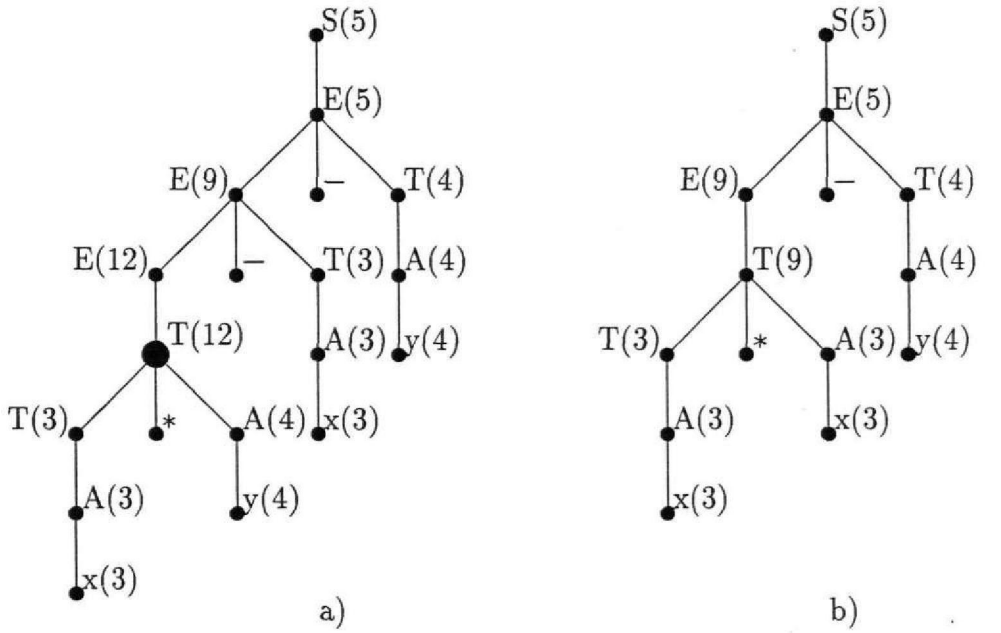


Figure 5: a) Unacceptable inference tree. b) Acceptable inference tree.

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 $+\cdot$:

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

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 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 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 6 *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 $G = (T, N, P, S)$:

- T — finite terminal symbol set;
- N — finite nonterminal symbol set;

- $S \in N$ — start symbol;
- P — a finite production set, where each production ($1 \leq k \leq |P|$) 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 $C_{i_0} \leftarrow C_{i_1} C_{i_2} \ldots C_{i_j}$, then the corresponding function is $f(c_{i_1}, \ldots, c_{i_j})$.

Additionally, every production $C_{i_0} \leftarrow C_{i_1} C_{i_2} \ldots C_{i_j}$ has a corresponding predicate $R_i : \{c_{i_0} \text{ domain set}\} \rightarrow \{True, False\}$ 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 $\langle C_{i_0}, K_{i_1}, K_{i_2} \ldots K_{i_j} \rangle$ that corresponds to the production $P_i = (C_{i_0} \leftarrow C_{i_1} C_{i_2} \ldots C_{i_j})$ has the same attribute as nonterminal C_{i_0}, and it is evaluated by first evaluating the values of attributes $c_{i_1}, \ldots, c_{i_j}$ that are assigned to $K_{i_1}, \ldots, K_{i_j}$ 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 $C_{i_0} \leftarrow C_{i_1} C_{i_2} \ldots C_{i_j}$ is defined by n equalities of the form $f(c_{i_1}, \ldots, c_{i_j}) = c_{i_0}$, 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 $\langle C_{i_k}, c_{i_k} \rangle$ $(1 \leq k \leq j)$ 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 $C_{i_0} \leftarrow C_{i_1} C_{i_2} \ldots C_{i_j}$, the weight of this node equals $\max(w_1, \ldots, w_j) + 1$, where w_i are

the 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 $w_1, \ldots, w_k$, 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).

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 $p_1, \ldots, p_k$ 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.

Lemma 7 *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 $\langle S, K_{i,1}, \ldots, K_{i,n} \rangle$, 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 $P \leftarrow S_1 \ldots S_n$, 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. 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 a breadth-first manner. A potentially infinite queue will be used for storing marked inference trees. By a *marked inference tree* we will understand an annotated inference tree 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 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 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 8 *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 complexity 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 8) is monotonic and the corresponding predicates are monotonic as well, the following theorem holds.

Theorem 9 *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 con-*

dition 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 of the previous section 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 6, i. e., $O(|w_i|)$ instead of $O(|w_i| \times \log |G_k|)$.

3.3 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 corre-

spondence rules by the following attribute grammar:

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

$$
\begin{array}{ll}
sqrtfun(0) = 0; & sqrtfun(1) = \text{“undefined”} \\
sqrtfun(2) = 1; & sqrtfun(3) = \text{“undefined”}
\end{array}
$$

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 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 corresponding 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 $map : N^*, (N \rightarrow N) \rightarrow N^*$ takes as arguments a string consisting of natural numbers and a function from N to N and returns a string over 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. $left, right : N^* \rightarrow N^*$ 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 $length : N^* \rightarrow N$ 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 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 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:

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

Similarly we can ensure that the statement has at least one occurrence of string variable w and that outer symbols of expressions on both sides of equality sign are different (to exclude enumeration of trivial statements like $left(w) = left(w)$)). According to Corollary 8 it is possible to efficiently enumerate all statements that conform to all these evaluation function and condition sets.

4 More complex attribute grammars

4.1 Introduction

In this section the question of whether more complex attribute grammars than considered in the previous section can be used for describing inductive synthesis search space will be studied. Let us shortly remind the problem statement. We want to be able to use *a priori* knowledge in the process of inductive synthesis, and we should be able to accommodate a wide range of possible types of such knowledge.

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 section 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? In the previous sections we already examined some possibilities; here we want to study a more general type of knowledge presentation.

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 is convenient to describe such syntactic objects by means of attribute grammars. Then a grammar generates a language with strings belonging to this language being our syntactic objects. Here the type of attribute grammars we are able to deal with will be more general than only attribute grammars with one synthesized attribute, as in the previous section. Extending the class of attribute grammars enables us to encode even wider range of hypothetical knowledge about the unknown object

The main problem we will solve here 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 section is to show that, if some conditions hold, it is possible.

The results discussed in this section generalize the results presented in the previous sections. We show that a more general class of attribute grammars can be considered, having possibly also inherited attributes and dependencies between different attributes. Thus by this section we conclude the research

direction this dissertation is devoted to.

4.2 Definitions

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:

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

In the previous section 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 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.

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 ($1 \leq k \leq |P|$) 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., $(c_{k,0}, \ldots, c_{k,j})$.

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 $C_{i_0} \leftarrow C_{i_1} C_{i_2} \ldots C_{i_j}$, then there is a corresponding function for each synthesized attribute of C_{i_0} and for each inherited attribute of $C_{i_1} C_{i_2} \ldots C_{i_j}$. For a synthesized attribute $c^{syn}_{i_0,k}$ the evaluation function is $f(\bar{c}_{i_0}, \ldots, \bar{c}_{i_j})$, 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 $\langle C_{i_0}, K_{i_1}, K_{i_2} \ldots K_{i_j} \rangle$ that corresponds to the production $P_i = (C_{i_0} \leftarrow C_{i_1} C_{i_2} \ldots C_{i_j})$ has the same synthesized attributes as nonterminal C_{i_0}, and attributes are evaluated by first evaluating the values of attributes $\bar{c}_{i_1}, \ldots, \bar{c}_{i_j}$ that are assigned to $K_{i_1}, \ldots, K_{i_j}$ 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^+.

4.3 The main result

Theorem 10 *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.*

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 consecutive 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 10 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 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.

4.4 Notes on implementation details

The described algorithms are under development as 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. Graph cleaning can help us to manipulate with more attributes and with deeper inference trees.

5 Conclusion

To summarize, the main result of this dissertation is construction of the following three algorithms:

- an algorithm that can efficiently enumerate a set of arithmetic expressions that correspond to several input/output examples and match some simple restrictions on syntactic structure;
- an algorithm that can efficiently enumerate a language generated by a simple attribute grammar with only synthesized attributes (and some other restrictions);
- an algorithm that can efficiently enumerate a language generated by a more general kind of attribute grammars.

These results were presented at several international conferences, including International Conferences on Algorithmic Learning Theory in 1993 (Tokyo), 1996 (Sydney) and 1998 (Otzenhausen, Germany).

Usually "inductive synthesis" means rigorous, but practically unimplementable algorithms, and "machine learning" means practical approach and

theoretically not so clear, *ad hoc* algorithms. Here we have tried to use rigorous algorithms to practical synthesis problems.

Although in some sense this dissertation concludes a series of related works, lots of unresolved and interesting problems remain, especially from the practical side. It is necessary to finish implementation of the most general version of the presented algorithm as a "workbench" of practical inductive synthesis, and to make extensive computer experiments.

References

[1] D.Angluin: Easily inferred sequences. Memorandum No. ERL-M499, University of California, 1974

[2] G.Barzdins: Inductive synthesis of term rewriting systems. Lecture Notes in Computer Science, vol.502 (1991), pp. 253–285

[3] J.M.Barzdins and G.J.Barzdins: Rapid construction of algebraic axioms from samples. Theoretical Computer Science 90 (1991), pp. 199–208

[4] J.Barzdins, G.Barzdins, K.Apsitis, U.Sarkans. Towards Efficient Inductive Synthesis of Expressions from input/output Examples. Lecture Notes in Computer Science, vol.744 (1993), pp. 59–72.

[5] J.Barzdins, U.Sarkans. Incorporating Hypothetical Knowledge into the Process of Inductive Synthesis. Lecture Notes in Computer Science, vol.1160 (1996), pp. 156–168.

[6] P.van Emde Boas, R.Kaas and E.Zijlstra: Design and Implementation of an Efficient Priority Queue. Mathematical Systems Theory 10 (1977), pp. 99–127

[7] J.Barzdins and G.Barzdins: Towards efficient inductive synthesis: rapid construction of local regularities. Lecture Notes in Computer Science, vol.659 (1993), pp. 132–140

[8] R.Freivalds: Inductive inference of recursive functions. Qualitative theory. Lecture Notes in Computer Science, vol.502 (1991), pp. 77–110

[9] R.Freivalds, J.Barzdins, K.Podnieks: Inductive inference of recursive functions: Complexity bounds. Lecture Notes in Computer Science, vol.502 (1991), pp. 565–613

[10] J.H.Holland. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. MIT Press, 1992.

[11] E.Howard: An Introduction to the History of Mathematics. New York: Holt, Reinhard and Winston 1961

[12] J.R.Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.

[13] J.R.Koza. Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, 1994.

[14] 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.

[15] P.Langley, H.A.Simon, G.Bradshaw. Heuristics for Empirical Discovery. In Computational Models of Learning, L.Bolc (ed.), Springer-Verlag, 1987.

[16] U.Sarkans, J.Barzdins. Using Attribute Grammars for Description of Inductive Inference Search Space. To appear in Proceedings of the 9th International Conference on Algorithmic Learning Theory, Otzenhausen, Germany, 1998.